

WORKING DRAFT

J3/07-007

5th January 2007 18:42

This is an internal working document of J3.

Contents

1	Overview	1
1.1	Scope	1
1.2	Processor	1
1.3	Inclusions	1
1.4	Exclusions	1
1.5	Conformance	2
1.6	Compatibility	3
1.6.1	New intrinsic procedures	3
1.6.2	New intrinsic data type and operator	3
1.6.3	Fortran 2003 compatibility	3
1.6.4	Fortran 95 compatibility	3
1.6.5	Fortran 90 compatibility	4
1.6.6	FORTTRAN 77 compatibility	4
1.7	Notation used in this part of ISO/IEC 1539	5
1.7.1	Applicability of requirements	5
1.7.2	Informative notes	5
1.7.3	Syntax rules	5
1.7.4	Constraints	6
1.7.5	Assumed syntax rules	6
1.7.6	Syntax conventions and characteristics	7
1.7.7	Text conventions	7
1.8	Deleted and obsolescent features	7
1.8.1	General	7
1.8.2	Nature of deleted features	7
1.8.3	Nature of obsolescent features	7
1.9	Normative references	8
2	Fortran terms and concepts	9
2.1	High level syntax	9
2.2	Program unit concepts	12
2.2.1	Program units and scoping units	12
2.2.2	Program	12
2.2.3	Main program	13
2.2.4	Procedure	13
2.2.5	Module	14
2.2.6	Submodule	14
2.3	Execution concepts	14
2.3.1	Statement classification	14
2.3.2	Program execution	14
2.3.3	Executable/nonexecutable statements	15
2.3.4	Statement order	15
2.3.5	The END statement	16
2.3.6	Execution sequence	16
2.4	Data concepts	17
2.4.1	Type	17
2.4.2	Data value	18
2.4.3	Data entity	18

2.4.4	Scalar	19
2.4.5	Array	19
2.4.6	Co-array	20
2.4.7	Pointer	20
2.4.8	Storage	21
2.5	Fundamental terms	21
2.5.1	Name and designator	21
2.5.2	Keyword	21
2.5.3	Association	22
2.5.4	Declaration	22
2.5.5	Definition	22
2.5.6	Reference	22
2.5.7	Intrinsic	22
2.5.8	Operator	23
2.5.9	Sequence	23
2.5.10	Companion processors	23
3	Lexical tokens, source form, and macro processing	25
3.1	Processor character set	25
3.1.1	Letters	25
3.1.2	Digits	25
3.1.3	Underscore	25
3.1.4	Special characters	26
3.1.5	Other characters	27
3.2	Low-level syntax	27
3.2.1	Names	27
3.2.2	Constants	28
3.2.3	Operators	28
3.2.4	Statement labels	29
3.2.5	Delimiters	30
3.3	Source form	30
3.3.1	Free source form	30
3.3.2	Fixed source form	33
3.4	Including source text	34
3.5	Macro processing	35
3.5.1	Macro definition	35
3.5.2	Macro expansion	37
4	Types	43
4.1	The concept of type	43
4.1.1	Set of values	43
4.1.2	Constants	43
4.1.3	Operations	43
4.2	Type parameters	43
4.3	Relationship of types and values to objects	45
4.3.1	Type specifiers and type compatibility	45
4.4	Intrinsic types	47
4.4.1	Classification and specification	47
4.4.2	Integer type	47
4.4.3	Real type	48
4.4.4	Complex type	50
4.4.5	Character type	51
4.4.6	Logical type	55
4.4.7	Bits type	55

4.5	Derived types	57
4.5.1	Derived type concepts	57
4.5.2	Derived-type definition	57
4.5.3	Derived-type parameters	61
4.5.4	Components	62
4.5.5	Type-bound procedures	70
4.5.6	Final subroutines	72
4.5.7	Type extension	74
4.5.8	Derived-type values	77
4.5.9	Derived-type specifier	77
4.5.10	Construction of derived-type values	77
4.5.11	Derived-type operations and assignment	80
4.6	Enumerations and enumerators	80
4.7	Construction of array values	81
5	Attribute declarations and specifications	85
5.1	General	85
5.2	Type declaration statements	85
5.2.1	Syntax	85
5.2.2	Automatic data objects	86
5.2.3	Initialization	87
5.2.4	Examples of type declaration statements	87
5.3	Attributes	87
5.3.1	Constraints	87
5.3.2	Accessibility attribute	88
5.3.3	ALLOCATABLE attribute	88
5.3.4	ASYNCHRONOUS attribute	88
5.3.5	BIND attribute for data entities	89
5.3.6	CONTIGUOUS attribute	89
5.3.7	DIMENSION attribute	90
5.3.8	EXTERNAL attribute	95
5.3.9	INTENT attribute	95
5.3.10	INTRINSIC attribute	97
5.3.11	OPTIONAL attribute	97
5.3.12	PARAMETER attribute	98
5.3.13	POINTER attribute	98
5.3.14	PROTECTED attribute	98
5.3.15	SAVE attribute	99
5.3.16	TARGET attribute	100
5.3.17	VALUE attribute	100
5.3.18	VOLATILE attribute	100
5.4	Attribute specification statements	101
5.4.1	Accessibility statement	101
5.4.2	ALLOCATABLE statement	102
5.4.3	ASYNCHRONOUS statement	102
5.4.4	BIND statement	102
5.4.5	CONTIGUOUS statement	102
5.4.6	DATA statement	102
5.4.7	DIMENSION statement	105
5.4.8	INTENT statement	105
5.4.9	OPTIONAL statement	106
5.4.10	PARAMETER statement	106
5.4.11	POINTER statement	106
5.4.12	PROTECTED statement	106

5.4.13	SAVE statement	107
5.4.14	TARGET statement	107
5.4.15	VALUE statement	107
5.4.16	VOLATILE statement	107
5.5	IMPLICIT statement	107
5.6	NAMelist statement	110
5.7	Storage association of data objects	110
5.7.1	EQUIVALENCE statement	110
5.7.2	COMMON statement	113
5.7.3	Restrictions on common and equivalence	115
6	Use of data objects	117
6.1	Scalars	118
6.1.1	Substrings	118
6.1.2	Structure components	118
6.1.3	Complex parts	120
6.1.4	Type parameter inquiry	121
6.2	Arrays	121
6.2.1	Whole arrays	121
6.2.2	Array elements and array sections	122
6.2.3	Image selectors	125
6.3	Dynamic association	125
6.3.1	ALLOCATE statement	125
6.3.2	NULLIFY statement	130
6.3.3	DEALLOCATE statement	130
6.3.4	STAT= specifier	132
6.3.5	ERRMSG= specifier	132
7	Expressions and assignment	133
7.1	Expressions	133
7.1.1	General	133
7.1.2	Form of an expression	133
7.1.3	Precedence of operators	137
7.1.4	Evaluation of operations	139
7.1.5	Intrinsic operations	140
7.1.6	Defined operations	150
7.1.7	Evaluation of operands	151
7.1.8	Integrity of parentheses	152
7.1.9	Type, type parameters, and shape of an expression	152
7.1.10	Conformability rules for elemental operations	154
7.1.11	Specification expression	154
7.1.12	Initialization expression	156
7.2	Assignment	157
7.2.1	Assignment statement	157
7.2.2	Pointer assignment	163
7.2.3	Masked array assignment – WHERE	166
7.2.4	FORALL	169
8	Execution control	177
8.1	Executable constructs containing blocks	177
8.1.1	General	177
8.1.2	Rules governing blocks	177
8.1.3	ASSOCIATE construct	178
8.1.4	BLOCK construct	179

8.1.5	CASE construct	180
8.1.6	CRITICAL construct	183
8.1.7	DO construct	184
8.1.8	IF construct and statement	190
8.1.9	SELECT TYPE construct	192
8.1.10	EXIT statement	194
8.2	Branching	195
8.2.1	GO TO statement	195
8.2.2	Computed GO TO statement	195
8.2.3	Arithmetic IF statement	195
8.3	CONTINUE statement	196
8.4	STOP statement	196
8.5	Image execution control	197
8.5.1	Image control statements	197
8.5.2	SYNC ALL statement	199
8.5.3	SYNC TEAM statement	200
8.5.4	SYNC IMAGES statement	201
8.5.5	NOTIFY and QUERY statements	203
8.5.6	SYNC MEMORY statement	204
8.5.7	STAT= and ERRMSG= specifiers in image execution control statements	206
9	Input/output statements	207
9.1	Records	207
9.1.1	Formatted record	207
9.1.2	Unformatted record	208
9.1.3	Endfile record	208
9.2	External files	208
9.2.1	File existence	209
9.2.2	File access	209
9.2.3	File position	211
9.2.4	File storage units	213
9.3	Internal files	214
9.4	File connection	214
9.4.1	Connection modes	215
9.4.2	Unit existence	215
9.4.3	Connection of a file to a unit	216
9.4.4	Preconnection	216
9.4.5	OPEN statement	217
9.4.6	CLOSE statement	222
9.5	Data transfer statements	223
9.5.1	General	223
9.5.2	Control information list	224
9.5.3	Data transfer input/output list	229
9.5.4	Execution of a data transfer input/output statement	231
9.5.5	Termination of data transfer statements	243
9.6	Waiting on pending data transfer	243
9.6.1	Wait operation	243
9.6.2	WAIT statement	244
9.7	File positioning statements	244
9.7.1	Syntax	244
9.7.2	BACKSPACE statement	245
9.7.3	ENDFILE statement	245
9.7.4	REWIND statement	246
9.8	FLUSH statement	246

9.9	File inquiry statement	247
9.9.1	Forms of the INQUIRE statement	247
9.9.2	Inquiry specifiers	248
9.9.3	Inquire by output list	254
9.10	Error, end-of-record, and end-of-file conditions	254
9.10.1	Error conditions and the ERR= specifier	255
9.10.2	End-of-file condition and the END= specifier	255
9.10.3	End-of-record condition and the EOR= specifier	255
9.10.4	IOSTAT= specifier	256
9.10.5	IOMSG= specifier	257
9.11	Restrictions on input/output statements	257
10	Input/output editing	259
10.1	Format specifications	259
10.2	Explicit format specification methods	259
10.2.1	FORMAT statement	259
10.2.2	Character format specification	259
10.3	Form of a format item list	260
10.3.1	Syntax	260
10.3.2	Edit descriptors	260
10.3.3	Fields	262
10.4	Interaction between input/output list and format	262
10.5	Positioning by format control	264
10.6	Decimal symbol	264
10.7	Data edit descriptors	264
10.7.1	General	264
10.7.2	Numeric and bits editing	265
10.7.3	Logical editing	271
10.7.4	Character editing	272
10.7.5	Generalized editing	272
10.7.6	User-defined derived-type editing	274
10.8	Control edit descriptors	274
10.8.1	Position editing	275
10.8.2	Slash editing	276
10.8.3	Colon editing	276
10.8.4	SS, SP, and S editing	276
10.8.5	P editing	276
10.8.6	BN and BZ editing	277
10.8.7	RU, RD, RZ, RN, RC, and RP editing	277
10.8.8	DC and DP editing	277
10.9	Character string edit descriptors	278
10.10	List-directed formatting	278
10.10.1	General	278
10.10.2	Values and value separators	278
10.10.3	List-directed input	279
10.10.4	List-directed output	281
10.11	Namelist formatting	282
10.11.1	General	282
10.11.2	Name-value subsequences	282
10.11.3	Namelist input	282
10.11.4	Namelist output	286
11	Program units	289
11.1	Main program	289

11.2	Modules	290
11.2.1	General	290
11.2.2	The USE statement and use association	291
11.2.3	Submodules	294
11.3	Block data program units	294
12	Procedures	297
12.1	Concepts	297
12.2	Procedure classifications	297
12.2.1	Procedure classification by reference	297
12.2.2	Procedure classification by means of definition	297
12.3	Characteristics	298
12.3.1	Characteristics of procedures	298
12.3.2	Characteristics of dummy arguments	298
12.3.3	Characteristics of function results	299
12.4	Procedure interface	299
12.4.1	General	299
12.4.2	Implicit and explicit interfaces	299
12.4.3	Specification of the procedure interface	300
12.5	Procedure reference	310
12.5.1	Syntax	310
12.5.2	Actual arguments, dummy arguments, and argument association	313
12.5.3	Function reference	325
12.5.4	Subroutine reference	325
12.5.5	Resolving named procedure references	326
12.5.6	Resolving type-bound procedure references	328
12.6	Procedure definition	328
12.6.1	Intrinsic procedure definition	328
12.6.2	Procedures defined by subprograms	329
12.6.3	Definition and invocation of procedures by means other than Fortran	336
12.6.4	Statement function	336
12.7	Pure procedures	337
12.8	Elemental procedures	339
12.8.1	Elemental procedure declaration and interface	339
12.8.2	Elemental function actual arguments and results	339
12.8.3	Elemental subroutine actual arguments	340
13	Intrinsic procedures and modules	341
13.1	Classes of intrinsic procedures	341
13.2	Arguments to intrinsic procedures	341
13.2.1	General rules	341
13.2.2	The shape of array arguments	342
13.2.3	Mask arguments	342
13.2.4	Arguments to collective subroutines	342
13.3	Bit model	342
13.4	Numeric models	343
13.5	Standard generic intrinsic procedures	344
13.5.1	Numeric functions	344
13.5.2	Mathematical functions	345
13.5.3	Character functions	345
13.5.4	Kind functions	346
13.5.5	Miscellaneous type conversion functions	346
13.5.6	Numeric inquiry functions	346
13.5.7	Array inquiry functions	346

13.5.8	Other inquiry functions	347
13.5.9	Bit manipulation procedures	347
13.5.10	Floating-point manipulation functions	347
13.5.11	Vector and matrix multiply functions	348
13.5.12	Array reduction functions	348
13.5.13	Array construction functions	348
13.5.14	Array location functions	348
13.5.15	Collective subroutines	349
13.5.16	Null function	349
13.5.17	Allocation transfer procedure	349
13.5.18	Random number subroutines	349
13.5.19	System environment procedures	349
13.6	Specific names for standard intrinsic functions	350
13.7	Specifications of the standard intrinsic procedures	351
13.8	Standard intrinsic modules	435
13.8.1	General	435
13.8.2	The ISO_FORTRAN_ENV intrinsic module	435
14	Exceptions and IEEE arithmetic	439
14.1	Derived types and constants defined in the modules	440
14.2	The exceptions	441
14.3	The rounding modes	443
14.4	Underflow mode	443
14.5	Halting	444
14.6	The floating-point status	444
14.7	Exceptional values	444
14.8	IEEE arithmetic	445
14.9	Tables of the procedures	445
14.9.1	Inquiry functions	445
14.9.2	Elemental functions	446
14.9.3	Kind function	446
14.9.4	Elemental subroutines	446
14.9.5	Nonelemental subroutines	447
14.10	Specifications of the procedures	447
14.11	Examples	462
15	Interoperability with C	467
15.1	General	467
15.2	The ISO_C_BINDING intrinsic module	467
15.2.1	Summary of contents	467
15.2.2	Named constants and derived types in the module	467
15.2.3	Procedures in the module	468
15.3	Interoperability between Fortran and C entities	472
15.3.1	General	472
15.3.2	Interoperability of intrinsic types	472
15.3.3	Interoperability with C pointer types	474
15.3.4	Interoperability of derived types and C struct types	475
15.3.5	Interoperability of scalar variables	476
15.3.6	Interoperability of array variables	476
15.3.7	Interoperability of procedures and procedure interfaces	477
15.4	Interoperation with C global variables	479
15.4.1	General	479
15.4.2	Binding labels for common blocks and variables	480
15.5	Interoperation with C functions	480

15.5.1	Definition and reference of interoperable procedures	480
15.5.2	Binding labels for procedures	481
15.5.3	Exceptions and IEEE arithmetic procedures	481
16	Scope, association, and definition	483
16.1	Identifiers and entities	483
16.2	Scope of global identifiers	483
16.3	Scope of local identifiers	484
16.3.1	Classes of local identifiers	484
16.3.2	Local identifiers that are the same as common block names	485
16.3.3	Function results	485
16.3.4	Components, type parameters, and bindings	485
16.3.5	Argument keywords	486
16.4	Statement and construct entities	486
16.5	Association	487
16.5.1	Name association	487
16.5.2	Pointer association	491
16.5.3	Storage association	493
16.5.4	Inheritance association	497
16.5.5	Establishing associations	497
16.6	Definition and undefinition of variables	497
16.6.1	Definition of objects and subobjects	497
16.6.2	Variables that are always defined	498
16.6.3	Variables that are initially defined	498
16.6.4	Variables that are initially undefined	498
16.6.5	Events that cause variables to become defined	498
16.6.6	Events that cause variables to become undefined	501
16.6.7	Variable definition context	503
16.6.8	Pointer association context	503
Annex A	(informative)Glossary of technical terms	505
Annex B	(informative)Decremental features	519
B.1	Deleted features	519
B.2	Obsolescent features	520
B.2.1	Alternate return	520
B.2.2	Computed GO TO statement	520
B.2.3	Statement functions	520
B.2.4	DATA statements among executables	521
B.2.5	Assumed character length functions	521
B.2.6	Fixed form source	521
B.2.7	CHARACTER* form of CHARACTER declaration	521
Annex C	(informative)Extended notes	523
C.1	Clause 4 notes	523
C.1.1	Selection of the approximation methods (4.4.3)	523
C.1.2	Type extension and component accessibility (4.5.2.2, 4.5.4)	523
C.1.3	Abstract types	524
C.1.4	Pointers (4.5.2)	525
C.1.5	Structure constructors and generic names	526
C.1.6	Generic type-bound procedures	528
C.1.7	Final subroutines (4.5.6, 4.5.6.2, 4.5.6.3, 4.5.6.4)	529
C.2	Clause 5 notes	531
C.2.1	The POINTER attribute (5.3.13)	531
C.2.2	The TARGET attribute (5.3.16)	532

	C.2.3	The VOLATILE attribute (5.3.18)	532
C.3	Clause 6	notes	533
	C.3.1	Structure components (6.1.2)	533
	C.3.2	Allocation with dynamic type (6.3.1)	535
	C.3.3	Pointer allocation and association	535
C.4	Clause 7	notes	536
	C.4.1	Character assignment	536
	C.4.2	Evaluation of function references	536
	C.4.3	Pointers in expressions	536
	C.4.4	Pointers on the left side of an assignment	537
	C.4.5	An example of a FORALL construct containing a WHERE construct	537
	C.4.6	Examples of FORALL statements	538
C.5	Clause 8	notes	539
	C.5.1	Loop control	539
	C.5.2	The CASE construct	539
	C.5.3	Examples of DO constructs	539
	C.5.4	Examples of invalid DO constructs	542
C.6	Clause 9	notes	543
	C.6.1	External files (9.2)	543
	C.6.2	Nonadvancing input/output (9.2.3.1)	544
	C.6.3	Asynchronous input/output	545
	C.6.4	OPEN statement (9.4.5)	546
	C.6.5	Connection properties (9.4.3)	548
	C.6.6	CLOSE statement (9.4.6)	548
C.7	Clause 10	notes	548
	C.7.1	Number of records (10.4, 10.5, 10.8.2)	548
	C.7.2	List-directed input (10.10.3)	549
C.8	Clause 11	notes	550
	C.8.1	Main program and block data program unit (11.1, 11.3)	550
	C.8.2	Dependent compilation (11.2)	550
	C.8.3	Examples of the use of modules	552
	C.8.4	Modules with submodules	558
C.9	Clause 12	notes	563
	C.9.1	Portability problems with external procedures (12.4.3.5)	563
	C.9.2	Procedures defined by means other than Fortran (12.6.3)	563
	C.9.3	Procedure interfaces (12.4)	564
	C.9.4	Abstract interfaces (12.4) and procedure pointer components (4.5)	564
	C.9.5	Argument association and evaluation (12.5.2)	566
	C.9.6	Pointers and targets as arguments (12.5.2.5, 12.5.2.7, 12.5.2.8)	567
	C.9.7	Polymorphic Argument Association (12.5.2.10)	568
C.10	Clause 13	notes	570
	C.10.1	Module for THIS_IMAGE and IMAGE_INDEX	570
	C.10.2	Collective co-array subroutine variations	570
C.11	Clause 15	notes	571
	C.11.1	Runtime environments	571
	C.11.2	Examples of Interoperation between Fortran and C Functions	571
C.12	Clause 16	notes	577
	C.12.1	Examples of host association (16.5.1.4)	577
	C.12.2	Rules ensuring unambiguous generics (12.4.3.4.5)	578
C.13	Array feature	notes	582
	C.13.1	Summary of features	582
	C.13.2	Examples	583
	C.13.3	FORMula TRANslation and array processing	588
	C.13.4	Sum of squared residuals	589

C.13.5	Vector norms: infinity-norm and one-norm	589
C.13.6	Matrix norms: infinity-norm and one-norm	589
C.13.7	Logical queries	589
C.13.8	Parallel computations	590
C.13.9	Example of element-by-element computation	590
C.13.10	Bit manipulation and inquiry procedures	591
Annex D	(informative)Processor Dependencies	593
D.1	Unspecified Items	593
D.2	Processor Dependencies	593
D.3	Unresolved Technical Issues	596
Annex E	(informative)Syntax rules	599
E.1	Extract of all syntax rules	599
E.2	Syntax rule cross-reference	644
Annex F	(informative)Index	659

List of Tables

2.1	Requirements on statement ordering	15
2.2	Statements allowed in scoping units	15
3.1	Special characters	27
6.1	Subscript order value	123
7.2	Categories of operations and relative precedence	137
7.3	Type of operands and results for intrinsic operators	141
7.4	Interpretation of the numeric intrinsic operators	143
7.6	Interpretation of the character intrinsic operator //	145
7.7	Interpretation of the logical intrinsic operators	146
7.8	The values of operations involving logical intrinsic operators	146
7.9	Interpretation of the bits intrinsic operators	147
7.10	The values of bits intrinsic operations other than //	147
7.11	Interpretation of the relational intrinsic operators	148
7.12	Type conformance for the intrinsic assignment statement	158
7.13	Numeric conversion and the assignment statement	160
7.14	Bits conversion and the assignment statement	161
10.1	E and D exponent forms	268
10.2	EN exponent forms	269
10.3	ES exponent forms	270
13.1	Characteristics of the result of NULL ()	412
15.1	Names of C characters with special semantics	468
15.2	Interoperability between Fortran and C types	472

Foreword

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

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

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

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

ISO/IEC 1539-1 was prepared by Joint Technical Committee ISO/IEC/JTC1, *Information technology*, Subcommittee SC22, *Programming languages, their environments and system software interfaces*.

This fifth edition cancels and replaces the fourth edition (ISO/IEC 1539-1:2004), which has been technically revised. It also incorporates the Technical Corrigenda ISO/IEC 1539-1:2004/Cor. 1:2005 and ISO/IEC 15391:2004/Cor. 2:2006.

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

- *Part 1: Base language*
- *Part 2: Varying length character strings*
- *Part 3: Conditional Compilation*

Introduction

International Standard programming language Fortran

This part of the International Standard comprises the specification of the base Fortran language, informally known as Fortran 2008. With the limitations noted in 1.6.3, the syntax and semantics of Fortran 2003 are contained entirely within Fortran 2008. Therefore, any standard-conforming Fortran 2003 program not affected by such limitations is a standard-conforming Fortran 2008 program. New features of Fortran 2008 can be compatibly incorporated into such Fortran 2003 programs, with any exceptions indicated in the text of this part of ISO/IEC 1539.

Fortran 2008 contains several extensions to Fortran 2003; some of these are listed below.

- (1) The maximum rank of an array has been increased from seven to fifteen.
- (2) Performance enhancements: The DO CONCURRENT construct, which allows loop iterations to be executed in any order or potentially concurrently.
- (3) Pointers can be initialized to point to a target.
- (4) Performance enhancements: CONTIGUOUS attribute.
- (5) The ATAN intrinsic is extended so that ATAN (Y, X) is ATAN2 (Y,X).
- (6) Allocatable components of recursive type.
- (7) The MOLD= specifier has been added to the ALLOCATE statement.
- (8) OPEN statement enhancements that allow the processor to select a unit number when opening an external unit. Such a unit number is guaranteed not to interfere with any program-managed unit numbers.
- (9) The BLOCK construct (allows declarations within executable statements).
- (10) A disassociated or deallocated actual argument can correspond to an optional nonpointer nonallocatable dummy argument.
- (11) The concept of *variable* now includes references to pointer functions which return associated pointers.
- (12) The COMPILER_VERSION and COMPILER_OPTIONS functions provide information about the translation phase of the execution of a program.
- (13) The real and imaginary parts of a COMPLEX variable can be selected using a component-like syntax.
- (14) Scoped macros which can generate whole Fortran statements and subprograms.
- (15) A FINDLOC intrinsic was added and a BACK= argument was also added to MAXLOC and MINLOC.
- (16) Parallel programming support: SPMD parallel programming, co-arrays for data exchange between images, image control statements, and collective procedures.
- (17) A BITS data type for non-numeric programming and enhanced handling of binary, octal, and hexadecimal constants.
- (18) The G0 edit descriptor.
- (19) Additional mathematical intrinsic functions for computing Bessel functions, error functions, the Gamma function, and generalized L_2 norms.

J3 internal note

Unresolved Technical Issue 080

The laundry list needs to be redone at a later time. RAH suggests going down the list of things in spread sheet and having a big feature and a little feature list.

Organization of this part of ISO/IEC 1539

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

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

It also contains the following nonnormative material:

Glossary	A
Decremental features	B
Extended notes	C
Syntax rules	D
Index	E

Information technology — Programming languages — Fortran —

Part 1: Base Language

1 Overview

1.1 Scope

ISO/IEC 1539 is a multipart International Standard; the parts are published separately. This publication, ISO/IEC 1539-1, which is the first part, specifies the form and establishes the interpretation of programs expressed in the base Fortran language. The purpose of this part of ISO/IEC 1539 is to promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on a variety of computing systems. The second part, ISO/IEC 1539-2, defines additional facilities for the manipulation of character strings of variable length; this has been largely subsumed by allocatable characters with deferred length parameters. The third part, ISO/IEC 1539-3, defines a standard conditional compilation facility for Fortran. A processor conforming to part 1 need not conform to ISO/IEC 1539-2 or ISO/IEC 1539-3; however, conformance to either assumes conformance to this part.

1.2 Processor

The combination of a computing system and the mechanism by which programs are transformed for use on that computing system is called a **processor** in this part of ISO/IEC 1539.

1.3 Inclusions

This part of ISO/IEC 1539 specifies

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

1.4 Exclusions

This part of ISO/IEC 1539 does not specify

- (1) the mechanism by which programs are transformed for use on computing systems,
- (2) the operations required for setup and control of the use of programs on computing systems,
- (3) the method of transcription of programs or their input or output data to or from a storage medium,
- (4) the program and processor behavior when this part of ISO/IEC 1539 fails to establish an interpretation except for the processor detection and reporting requirements in items (2) to (8) of 1.5,

- 1 (5) the size or complexity of a program and its data that will exceed the capacity of any
- 2 particular computing system or the capability of a particular processor,
- 3 (6) the mechanism for determining the number of images of a program,
- 4 (7) the physical properties of an image or the relationship between images and the computational
- 5 elements of a computing system,

J3 internal note

Unresolved Technical Issue 099

Image properties not specified by the standard.

According to later notes, we don't specify this kind of stuff.

I think it belongs in Exclusions. It *definitely* does not belong in a note!

- 6 (8) the physical properties of the representation of quantities and the method of rounding,
- 7 approximating, or computing numeric values on a particular processor,
- 8 (9) the physical properties of input/output records, files, and units, or
- 9 (10) the physical properties and implementation of storage.

1.5 Conformance

A program (2.2.2) is a **standard-conforming program** if it uses only those forms and relationships described herein and if the program has an interpretation according to this part of ISO/IEC 1539. A program unit (2.2.1) conforms to this part of ISO/IEC 1539 if it can be included in a program in a manner that allows the program to be standard conforming.

A processor conforms to this part of ISO/IEC 1539 if:

- (1) it executes any standard-conforming program in a manner that fulfills the interpretations herein, subject to any limits that the processor may impose on the size and complexity of the program;
- (2) it contains the capability to detect and report the use within a submitted program unit of a form designated herein as obsolescent, insofar as such use can be detected by reference to the numbered syntax rules and constraints;
- (3) it contains the capability to detect and report the use within a submitted program unit of an additional form or relationship that is not permitted by the numbered syntax rules or constraints, including the deleted features described in Annex B;
- (4) it contains the capability to detect and report the use within a submitted program unit of an intrinsic type with a kind type parameter value not supported by the processor (4.4);
- (5) it contains the capability to detect and report the use within a submitted program unit of source form or characters not permitted by Clause 3;
- (6) it contains the capability to detect and report the use within a submitted program of name usage not consistent with the scope rules for names, labels, operators, and assignment symbols in Clause 16;
- (7) it contains the capability to detect and report the use within a submitted program unit of intrinsic procedures whose names are not defined in Clause 13; and
- (8) it contains the capability to detect and report the reason for rejecting a submitted program.

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

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

1 name of a procedure in a standard-conforming program. If such a conflict occurs and involves the name
2 of an external procedure, the processor is permitted to use the intrinsic procedure unless the name is
3 given the EXTERNAL attribute (5.3.8) in the scoping unit (2.2.1). A standard-conforming program
4 shall not use nonstandard intrinsic procedures or modules that have been added by the processor.

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

9 In some cases, the provision of facilities that are not completely specified in this standard is allowed.
10 These facilities are identified as **processor dependent**. They shall be provided, with methods or
11 semantics determined by the processor.

NOTE 1.1

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

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

12 1.6 Compatibility

13 1.6.1 New intrinsic procedures

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

19 1.6.2 New intrinsic data type and operator

20 This part of ISO/IEC 1539 specifies a new intrinsic type, BITS, which will conflict with a derived type
21 of the same name.

22 This part of ISO/IEC 1539 specifies a new intrinsic operator, .XOR., which might conflict with a user-
23 defined operator of the same name, has a different precedence from that of a user-defined operator, and
24 a different syntax from that of a user-defined unary operator.

25 1.6.3 Fortran 2003 compatibility

26 Except as identified in this subclause, this part of ISO/IEC 1539 is an upward compatible extension
27 to the preceding Fortran International Standard, ISO/IEC 1539-1:2004 (Fortran 2003). Any standard-
28 conforming Fortran 2003 program that does not use a derived type called BITS, and does not use a
29 user-defined operator called .XOR., remains standard-conforming under this part of ISO/IEC 1539.

30 1.6.4 Fortran 95 compatibility

31 Except as identified in this subclause, this part of ISO/IEC 1539 is an upward compatible extension to
32 ISO/IEC 1539-1:1997 (Fortran 95). Any standard-conforming Fortran 95 program that does not use a

derived type called BITS or a user-defined operator called .XOR. remains standard-conforming under this part of ISO/IEC 1539. The following Fortran 95 features may have different interpretations in this part of ISO/IEC 1539.

- (1) Earlier Fortran standards had the concept of printing, meaning that column one of formatted output had special meaning for a processor-dependent (possibly empty) set of external files. This could be neither detected nor specified by a standard-specified means. The interpretation of the first column is not specified by this part of ISO/IEC 1539.
- (2) This part of ISO/IEC 1539 specifies a different output format for real zero values in list-directed and namelist output.
- (3) If the processor can distinguish between positive and negative real zero, this part of ISO/IEC 1539 requires different returned values for ATAN2(Y,X) when $X < 0$ and Y is negative real zero and for LOG(X) and SQRT(X) when X is complex with $\text{REAL}(X) < 0$ and negative zero imaginary part.

1.6.5 Fortran 90 compatibility

Except for the deleted features noted in Annex B.1, and except as identified in this subclause, this part of ISO/IEC 1539 is an upward compatible extension to ISO/IEC 1539:1991 (Fortran 90). Any standard-conforming Fortran 90 program that does not use a derived type called BITS, a user-defined operator called .XOR., or one of the deleted features remains standard-conforming under this part of ISO/IEC 1539.

The PAD= specifier in the INQUIRE statement in this part of ISO/IEC 1539 returns the value UNDEFINED if there is no connection or the connection is for unformatted input/output. Fortran 90 specified YES.

Fortran 90 specified that if the second argument to MOD or MODULO was zero, the result was processor dependent. this part of ISO/IEC 1539 specifies that the second argument shall not be zero.

1.6.6 FORTRAN 77 compatibility

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

- (1) FORTRAN 77 permitted a processor to supply more precision derived from a real constant than can be represented in a real datum when the constant is used to initialize a data object of type double precision real in a DATA statement. This part of ISO/IEC 1539 does not permit a processor this option.
- (2) If a named variable that was not in a common block was initialized in a DATA statement and did not have the SAVE attribute specified, FORTRAN 77 left its SAVE attribute processor dependent. This part of ISO/IEC 1539 specifies (5.4.6) that this named variable has the SAVE attribute.
- (3) FORTRAN 77 specified that the number of characters required by the input list was to be less than or equal to the number of characters in the record during formatted input. This part of ISO/IEC 1539 specifies (9.5.4.4.2) 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.

- 1 (4) A value of 0 for a list item in a formatted output statement will be formatted in a different
 2 form for some G edit descriptors. In addition, this part of ISO/IEC 1539 specifies how
 3 rounding of values will affect the output field form, but FORTRAN 77 did not address this
 4 issue. Therefore, some FORTRAN 77 processors may produce an output form different from
 5 the output form produced by Fortran 2003 processors for certain combinations of values and
 6 G edit descriptors.
- 7 (5) If the processor can distinguish between positive and negative real zero, the behavior of the
 8 SIGN intrinsic function when the second argument is negative real zero is changed by this
 9 part of ISO/IEC 1539.

10 1.7 Notation used in this part of ISO/IEC 1539

11 1.7.1 Applicability of requirements

12 In this part of ISO/IEC 1539, “shall” is to be interpreted as a requirement; conversely, “shall not” is
 13 to be interpreted as a prohibition. Except where stated otherwise, such requirements and prohibitions
 14 apply to programs rather than processors.

15 1.7.2 Informative notes

16 Informative notes of explanation, rationale, examples, and other material are interspersed with the
 17 normative body of this part of ISO/IEC 1539. The informative material is nonnormative; it is identified
 18 by being in shaded, framed boxes that have numbered headings beginning with “NOTE.”

19 1.7.3 Syntax rules

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

- 22 (1) Characters from the Fortran character set (3.1) are interpreted literally as shown, except
 23 where otherwise noted.
- 24 (2) Lower-case italicized letters and words (often hyphenated and abbreviated) represent gen-
 25 eral syntactic classes for which particular syntactic entities shall be substituted in actual
 26 statements.

27 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

- 28 (3) 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

- 29 (4) Each syntax rule is given a unique identifying number of the form Rsnn, where s is a
 30 one- or two-digit clause number and nn is a two-digit sequence number within that clause.

- 1 The syntax rules are distributed as appropriate throughout the text, and are referenced by
 2 number as needed. Some rules in Clauses 2 and 3 are more fully described in later clauses; in
 3 such cases, the clause number *s* is the number of the later clause where the rule is repeated.
 4 (5) The syntax rules are not a complete and accurate syntax description of Fortran, and cannot
 5 be used to generate a Fortran parser automatically; where a syntax rule is incomplete, it is
 6 restricted by corresponding constraints and text.

NOTE 1.2

An example of the use of the syntax rules is:

digit-string **is** *digit* [*digit*] ...

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

digit
digit digit
digit digit digit digit
digit digit digit digit digit digit digit digit

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

4
 67
 1999
 10243852

7 **1.7.4 Constraints**

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

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

- 14 Some constraints are not associated with particular syntax rules. The effect of such a constraint is similar
 15 to that of a restriction stated in the text, except that a processor is required to have the capability to
 16 detect and report violations of constraints (1.5). In some cases, a broad requirement is stated in text
 17 and a subset of the same requirement is also stated as a constraint. This indicates that a standard-
 18 conforming program is required to adhere to the broad requirement, but that a standard-conforming
 19 processor is required only to have the capability of diagnosing violations of the constraint.

20 **1.7.5 Assumed syntax rules**

- 21 In order to minimize the number of additional syntax rules and convey appropriate constraint informa-
 22 tion, the following rules are assumed.

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

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

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

26 C101 (R103) *scalar-xyz* shall be scalar.

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

3 1.7.6 Syntax conventions and characteristics

- 4 (1) Any syntactic class name ending in “-*stmt*” follows the source form statement rules: it shall
 5 be delimited by end-of-line or semicolon, and may be labeled unless it forms part of another
 6 statement (such as an IF or WHERE statement). Conversely, everything considered to be
 7 a source form statement is given a “-*stmt*” ending in the syntax rules.
- 8 (2) The rules on statement ordering are described rigorously in the definition of *program-unit*
 9 (*R202*). Expression hierarchy is described rigorously in the definition of *expr* (*R722*).
- 10 (3) The suffix “-*spec*” is used consistently for specifiers, such as input/output statement speci-
 11 fiers. It also is used for type declaration attribute specifications (for example, “*array-spec*”
 12 in *R510*), and in a few other cases.
- 13 (4) Where reference is made to a type parameter, including the surrounding parentheses, the
 14 suffix “-*selector*” is used. See, for example, “*kind-selector*” (*R405*) and “*length-selector*”
 15 (*R421*).
- 16 (5) The term “*subscript*” (for example, *R619*, *R620*, and *R621*) is used consistently in array
 17 definitions.

18 1.7.7 Text conventions

19 In the descriptive text, an equivalent English word is frequently used in place of a syntactic term.
 20 Particular statements and attributes are identified in the text by an upper-case keyword, e.g., “END
 21 statement”. Boldface words are used in the text where they are first defined with a specialized meaning.
 22 The descriptions of obsolescent features appear in a smaller type size.

NOTE 1.3

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

23 1.8 Deleted and obsolescent features

24 1.8.1 General

25 This part of ISO/IEC 1539 protects the users’ investment in existing software by including all but five
 26 of the language elements of Fortran 90 that are not processor dependent. This part of ISO/IEC 1539
 27 identifies two categories of outmoded features. There are five in the first category, **deleted features**,
 28 which consists of features considered to have been redundant in FORTRAN 77 and largely unused in
 29 Fortran 90. Those in the second category, **obsolescent features**, are considered to have been redundant
 30 in Fortran 90 and Fortran 95, but are still frequently used.

31 1.8.2 Nature of deleted features

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

34 1.8.3 Nature of obsolescent features

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

37 The obsolescent features are identified in the text of this part of ISO/IEC 1539 by a distinguishing type
 38 font (1.7.7).

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

3 **1.9 Normative references**

- 4 The following referenced standards are indispensable for the application of this part of ISO/IEC 1539.
5 For dated references, only the edition cited applies. For undated references, the latest edition of the
6 referenced standard (including any amendments) applies.

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

8 ISO 8601:1988, *Data elements and interchange formats—Information interchange—*
9 *Representation of dates and times*.

10 ISO/IEC 9899:1999, *Information technology—Programming languages—C*.

11 ISO/IEC 10646-1:2000, *Information technology—Universal multiple-octet coded character set (UCS)—*
12 *Part 1: Architecture and basic multilingual plane*.

13 IEC 60559 (1989-01), *Binary floating-point arithmetic for microprocessor systems*.

14 ISO/IEC 646:1991 (International Reference Version) is the international equivalent of ANSI X3.4-1986,
15 commonly known as ASCII.

16 This part of ISO/IEC 1539 refers to ISO/IEC 9899:1999 as the C International Standard.

17 Because IEC 60559 (1989-01) was originally IEEE 754-1985, *Standard for binary floating-point arith-*
18 *metic*, and is widely known by this name, this part of ISO/IEC 1539 refers to it as the IEEE International
19 Standard.

2 Fortran terms and concepts

2.1 High level syntax

This clause introduces the terms associated with program units and other Fortran concepts above the construct, statement, and expression levels and illustrates their relationships. The notation used in this part of ISO/IEC 1539 is described in 1.7.

NOTE 2.1

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

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

A *program* shall contain exactly one *main-program program-unit* or a main program defined by means other than Fortran, but not both.

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

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

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

R1227 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1233 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-subroutine-stmt

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

R1116 *submodule* **is** *submodule-stmt*

1			[<i>specification-part</i>]
2			[<i>module-subprogram-part</i>]
3			<i>end-submodule-stmt</i>
4	R1120	<i>block-data</i>	is <i>block-data-stmt</i>
5			[<i>specification-part</i>]
6			<i>end-block-data-stmt</i>
7	R204	<i>specification-part</i>	is [<i>use-stmt</i>] ...
8			[<i>import-stmt</i>] ...
9			[<i>implicit-part</i>]
10			[<i>declaration-construct</i>] ...
11	R205	<i>implicit-part</i>	is [<i>implicit-part-stmt</i>] ...
12			<i>implicit-stmt</i>
13	R206	<i>implicit-part-stmt</i>	is <i>implicit-stmt</i>
14			or <i>parameter-stmt</i>
15			or <i>format-stmt</i>
16			or <i>entry-stmt</i>
17	R207	<i>declaration-construct</i>	is <i>derived-type-def</i>
18			or <i>entry-stmt</i>
19			or <i>enum-def</i>
20			or <i>format-stmt</i>
21			or <i>interface-block</i>
22			or <i>macro-definition</i>
23			or <i>parameter-stmt</i>
24			or <i>procedure-declaration-stmt</i>
25			or <i>specification-stmt</i>
26			or <i>type-declaration-stmt</i>
27			or <i>stmt-function-stmt</i>
28	R208	<i>execution-part</i>	is <i>executable-construct</i>
29			[<i>execution-part-construct</i>] ...
30	R209	<i>execution-part-construct</i>	is <i>executable-construct</i>
31			or <i>format-stmt</i>
32			or <i>entry-stmt</i>
33			or <i>data-stmt</i>
34	R210	<i>internal-subprogram-part</i>	is <i>contains-stmt</i>
35			[<i>internal-subprogram</i>] ...
36	R211	<i>internal-subprogram</i>	is <i>function-subprogram</i>
37			or <i>subroutine-subprogram</i>
38	R1107	<i>module-subprogram-part</i>	is <i>contains-stmt</i>
39			[<i>module-subprogram</i>] ...
40	R1108	<i>module-subprogram</i>	is <i>function-subprogram</i>
41			or <i>subroutine-subprogram</i>
42			or <i>separate-module-subprogram</i>
43	R1237	<i>separate-module-subprogram</i>	is <i>mp-subprogram-stmt</i>
44			[<i>specification-part</i>]
45			[<i>execution-part</i>]

1		[<i>internal-subprogram-part</i>]
2		<i>end-mp-subprogram-stmt</i>
3	R212	<i>specification-stmt</i>
4		is <i>access-stmt</i>
5		or <i>allocatable-stmt</i>
6		or <i>asynchronous-stmt</i>
7		or <i>bind-stmt</i>
8		or <i>common-stmt</i>
9		or <i>data-stmt</i>
10		or <i>dimension-stmt</i>
11		or <i>equivalence-stmt</i>
12		or <i>external-stmt</i>
13		or <i>intent-stmt</i>
14		or <i>intrinsic-stmt</i>
15		or <i>namelist-stmt</i>
16		or <i>optional-stmt</i>
17		or <i>pointer-stmt</i>
18		or <i>protected-stmt</i>
19		or <i>save-stmt</i>
20		or <i>target-stmt</i>
21		or <i>volatile-stmt</i>
22	R213	<i>executable-construct</i>
23		is <i>action-stmt</i>
24		or <i>associate-construct</i>
25		or <i>block-construct</i>
26		or <i>case-construct</i>
27		or <i>critical-construct</i>
28		or <i>do-construct</i>
29		or <i>forall-construct</i>
30		or <i>if-construct</i>
31		or <i>select-type-construct</i>
32	R214	<i>action-stmt</i>
33		is <i>allocate-stmt</i>
34		or <i>assignment-stmt</i>
35		or <i>backspace-stmt</i>
36		or <i>call-stmt</i>
37		or <i>close-stmt</i>
38		or <i>continue-stmt</i>
39		or <i>cycle-stmt</i>
40		or <i>deallocate-stmt</i>
41		or <i>end-function-stmt</i>
42		or <i>end-mp-subprogram-stmt</i>
43		or <i>end-program-stmt</i>
44		or <i>end-subroutine-stmt</i>
45		or <i>endfile-stmt</i>
46		or <i>exit-stmt</i>
47		or <i>flush-stmt</i>
48		or <i>forall-stmt</i>
49		or <i>goto-stmt</i>
50		or <i>if-stmt</i>
51		or <i>inquire-stmt</i>
52		or <i>notify-stmt</i>
		or <i>nullify-stmt</i>

1 or *open-stmt*
 2 or *pointer-assignment-stmt*
 3 or *print-stmt*
 4 or *query-stmt*
 5 or *read-stmt*
 6 or *return-stmt*
 7 or *rewind-stmt*
 8 or *stop-stmt*
 9 or *sync-all-stmt*
 10 or *sync-images-stmt*
 11 or *sync-memory-stmt*
 12 or *sync-team-stmt*
 13 or *wait-stmt*
 14 or *where-stmt*
 15 or *write-stmt*
 16 or *arithmetic-if-stmt*
 17 or *computed-goto-stmt*

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

20 Additionally, an EXPAND statement may occur anywhere that any statement may occur other than
 21 as the first statement of a program unit. The syntax rules are applied to the program after macro
 22 expansion, i.e. with each EXPAND statement replaced by the statements it produces.

23 2.2 Program unit concepts

24 2.2.1 Program units and scoping units

25 Program units are the fundamental components of a Fortran program. A **program unit** may be
 26 a main program, an external subprogram, a module, a submodule, or a block data program unit. A
 27 **subprogram** may be a function subprogram or a subroutine subprogram. A module contains definitions
 28 that are to be made accessible to other program units. A submodule is an extension of a module; it may
 29 contain the definitions of procedures declared in a module or another submodule. A block data program
 30 unit is used to specify initial values for data objects in named common blocks. Each type of program
 31 unit is described in Clause 11 or 12. An **external subprogram** is a subprogram that is not in a main
 32 program, a module, a submodule, or another subprogram. An **internal subprogram** is a subprogram
 33 that is in a main program or another subprogram. A **module subprogram** is a subprogram that is in
 34 a module or submodule but is not an internal subprogram.

35 A program unit consists of a set of nonoverlapping scoping units. A **scoping unit** is

- 36 (1) a program unit or subprogram, excluding any scoping units in it,
- 37 (2) a derived-type definition (4.5.2), or
- 38 (3) an interface body, excluding any scoping units in it.

39 A scoping unit that immediately surrounds another scoping unit is called the **host scoping unit** (often
 40 abbreviated to **host**). A module or submodule is also the host scoping unit of its child submodules.

41 2.2.2 Program

42 A **program** consists of exactly one main program, any number (including zero) of other kinds of program
 43 units, any number (including zero) of external procedures, and any number (including zero) of other
 44 entities defined by means other than Fortran.

NOTE 2.2

There is a restriction that there shall be no more than one unnamed block data program unit (11.3).

This part of ISO/IEC 1539 places no ordering requirement on the program units that constitute a program, but because the public portions of a module are required to be available by the time a module reference (11.2.2) is processed, a processor may require a particular order of processing of the program units.

2.2.3 Main program

The Fortran main program is described in 11.1.

2.2.4 Procedure

A **procedure** encapsulates an arbitrary sequence of actions that may be invoked directly during program execution. Procedures are either functions or subroutines. A **function** is a procedure that is invoked in an expression; its invocation causes a value to be computed which is then used in evaluating the expression. The variable that returns the value of a function is called the **result variable**. A **subroutine** is a procedure that is invoked in a CALL statement, by a defined assignment statement, or by some operations on derived-type entities. Unless it is a pure procedure, a subroutine may be used to change the program state by changing the values of any of the data objects accessible to the subroutine; unless it is a pure procedure, a function may do this in addition to computing the function value.

Procedures are described further in Clause 12.

2.2.4.1 External procedure

An **external procedure** is a procedure that is defined by an external subprogram or by means other than Fortran. An external procedure may be invoked by the main program or by any procedure of a program.

2.2.4.2 Module procedure

A **module procedure** is a procedure that is defined by a module subprogram (R1108). The module or submodule containing the subprogram is the **host** scoping unit of the module procedure.

2.2.4.3 Internal procedure

An **internal procedure** is a procedure that is defined by an internal subprogram (R211). The containing main program or subprogram is the **host** scoping unit of the internal procedure. An internal procedure is local to its host in the sense that its name is accessible within the host scoping unit and all its other internal procedures but is not accessible elsewhere.

2.2.4.4 Interface block

An **interface body** describes an abstract interface or the interface of a dummy procedure, external procedure, procedure pointer, or separate module procedure.

An **interface block** is a specific interface block, an abstract interface block, or a generic interface block. A specific interface block is a collection of interface bodies. A generic interface block can also be used to specify that a procedure can be invoked

- (1) by using a generic name,
- (2) by using a defined operator,

- 1 (3) by using a defined assignment, or
2 (4) for derived-type input/output.

3 2.2.5 Module

4 A **module** contains (or accesses from other modules) definitions that are to be made accessible to other
5 program units. These definitions include data object declarations, type definitions, procedure definitions,
6 and interface blocks. A scoping unit in another program unit may access the definitions in a module.
7 Modules are further described in Clause 11.

8 2.2.6 Submodule

9 A **submodule** is a program unit that extends a module or another submodule. It may provide definitions
10 (12.6) for procedures whose interfaces are declared (12.4.3.2) in an ancestor module or submodule. It
11 may also contain declarations and definitions of other entities, which are accessible in its descendants.
12 An entity declared in a submodule is not accessible by use association unless it is a module procedure
13 whose interface is declared in the ancestor module. Submodules are further described in Clause 11.

NOTE 2.3

The scoping unit of a submodule accesses the scoping unit of its parent module or submodule by host association.

14 2.3 Execution concepts

15 2.3.1 Statement classification

16 Each Fortran statement is classified as either an executable statement or a nonexecutable statement.
17 There are restrictions on the order in which statements may appear in a program unit, and not all
18 executable statements may appear in all contexts.

19 2.3.2 Program execution

20 An instance of a Fortran program is an **image**. Execution of a program consists of the asynchronous
21 execution of a fixed number (which may be one) of its images. Each image has its own execution state,
22 floating-point status (14.6), and set of data objects and procedure pointers. Whether a file is available
23 on any image or only on a specific image is processor dependent. Each image is identified by an **image**
24 **index**, which is an integer value in the range one to the number of images.

NOTE 2.4

The programmer controls the progress of execution in each image through explicit use of Fortran control constructs (8.1, 8.2). Image control statements (8.5.1) affect the relative progress of execution between images. Co-arrays (2.4.6) provide a mechanism for accessing data on one image from another image.

NOTE 2.5

A processor might allow the number of images to be chosen at compile time, link time, or run time. It might be the same as the number of CPUs but this is not required. Compiling for a single image might permit the optimizer to eliminate overhead associated with parallel execution. Portable programs should not make assumptions about the exact number of images. The maximum number of images may be limited due to architectural constraints.

25 A **team** is a set of images identified by a scalar object of type `IMAGE_TEAM` (13.8.2.7).

2.3.3 Executable/nonexecutable statements

Image execution is a sequence, in time, of actions. An **executable statement** is an instruction to perform or control one or more of these actions. Thus, the executable statements of a program unit determine the behavior of the program unit. The executable statements are all of those that make up the syntactic class *executable-construct*, excluding those in the *specification-part* of a BLOCK executable construct.

Nonexecutable statements do not specify actions; they are used to configure the program environment in which actions take place. The nonexecutable statements are all those not classified as executable.

2.3.4 Statement order

The syntax rules of clause 2.1 specify the statement order within program units and subprograms. These rules are illustrated in Table 2.1 and Table 2.2. Table 2.1 shows the ordering rules for statements and applies to all program units, subprograms, and interface bodies. Vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that shall not be interspersed. Internal or module subprograms shall follow a CONTAINS statement. Between USE and CONTAINS statements in a subprogram, nonexecutable statements generally precede executable statements, although the ENTRY statement, FORMAT statement, and DATA statement may appear among the executable statements. Table 2.2 shows which statements are allowed in a scoping unit.

Table 2.1: 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	Derived-type definitions, interface blocks, type declaration statements, enumeration definitions, procedure declarations, specification statements, and statement function statements
	DATA statements	Executable constructs
CONTAINS statement		
Internal subprograms or module subprograms		
END statement		

Table 2.2: Statements allowed in scoping units

Statement type	Kind of scoping unit ¹						
	Main program	Module or submodule	Block data	External subprog	Module subprog	Internal subprog	Interface body
USE	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IMPORT	No	No	No	No	No	No	Yes

Statements allowed in scoping units

Statement type	Kind of scoping unit ¹						
	Main program	Module or submodule	Block data	External subprog	Module subprog	Internal subprog	Interface body
ENTRY	No	No	No	Yes	Yes	No	No
FORMAT	Yes	No	No	Yes	Yes	Yes	No
Misc. decl.s ²	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DATA	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	Yes	Yes	No	Yes	Yes	Yes	Yes
Executable	Yes	No	No	Yes	Yes	Yes	No
CONTAINS	Yes	Yes	No	Yes	Yes	No	No
Statement function	Yes	No	No	Yes	Yes	Yes	No
(1) The scoping unit of a module or submodule does not include any module subprograms that it contains.							
(2) Miscellaneous declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, enumeration definitions, procedure declaration statements, and specification statements.							

2.3.5 The END statement

An *end-program-stmt*, *end-function-stmt*, *end-subroutine-stmt*, *end-mp-subprogram-stmt*, *end-module-stmt*, *end-submodule-stmt*, or *end-block-data-stmt* is an **END statement**. Each program unit, module subprogram, and internal subprogram shall have exactly one END statement. The *end-program-stmt*, *end-function-stmt*, *end-subroutine-stmt*, and *end-mp-subprogram-stmt* statements are executable, and may be branch target statements (8.2). Executing an *end-program-stmt* causes normal termination of execution of the program. Executing an *end-function-stmt*, *end-subroutine-stmt*, or *end-mp-subprogram-stmt* is equivalent to executing a *return-stmt* with no *scalar-int-expr*.

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

2.3.6 Execution sequence

If a program contains a Fortran main program, execution of the program begins by creating a fixed number of instances of the 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, RETURN, or END statement is executed.

- (1) Execution of a branching statement (8.2) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence.
- (2) CASE constructs, DO constructs, IF constructs, and SELECT TYPE constructs contain an internal statement structure and execution of these constructs involves implicit internal branching. See Clause 8 for the detailed semantics of each of these constructs.
- (3) BLOCK constructs may contain specification expressions; see 8.1.4 for detailed semantics of this construct.

1 (4) END=, ERR=, and EOR= specifiers may result in a branch.

2 (5) Alternate returns may result in a branch.

3 Internal subprograms may precede the END statement of a main program or a subprogram. The
4 execution sequence excludes all such definitions.

5 The relative ordering of the execution sequences of different images can be affected by image control
6 statements (8.5.1).

7 Normal termination of execution of an image occurs if a STOP statement is executed on that image. Ex-
8 ecution of an *end-program-stmt* results in a synchronization of all images followed by normal termination
9 of execution of all images. Normal termination of execution of an image also may occur during execution
10 of a procedure defined by a companion processor (C International Standard 5.1.2.2.3 and 7.20.4.3). If
11 normal termination of execution occurs within a Fortran program unit and the program incorporates
12 procedures defined by a companion processor, the process of execution termination shall include the
13 effect of executing the C exit() function (C International Standard 7.20.4.3).

14 2.4 Data concepts

15 Nonexecutable statements are used to specify the characteristics of the data environment. This includes
16 typing variables, declaring arrays, and defining new types.

17 2.4.1 Type

18 A **type** is a named category of data that is characterized by a set of values, a syntax for denoting
19 these values, and a set of operations that interpret and manipulate the values. This central concept is
20 described in 4.1.

21 A type may be parameterized, in which case the set of data values, the syntax for denoting them, and
22 the set of operations depend on the values of one or more parameters. Such a parameter is called a **type**
23 **parameter** (4.2).

24 There are two categories of types: intrinsic types and derived types.

25 2.4.1.1 Intrinsic type

26 An **intrinsic type** is a type that is defined by the language, along with operations, and is always
27 accessible. The intrinsic types are integer, real, complex, character, logical, and bits. The properties of
28 intrinsic types are described in 4.4. The intrinsic type parameters are KIND and LEN.

29 The **kind type parameter** indicates the representation method for the specified type.

30 2.4.1.2 Derived type

31 A **derived type** is a type that is defined by a type definition or by an intrinsic module. A scalar object of
32 derived type is called a **structure** (4.5). Derived types may be parameterized. Assignment of structures
33 is defined intrinsically (7.2.1.3), but there are no intrinsic operations for structures. For each derived
34 type, a structure constructor is available to provide values (4.5.10). In addition, data objects of derived
35 type may be used as procedure arguments and function results, and may appear in input/output lists.
36 If additional operations are needed for a derived type, they shall be supplied as procedure definitions.

37 Derived types are described further in 4.5.

38 2.4.2 Data value

1 Each intrinsic type has associated with it a set of values that a datum of that type may take, depending
 2 on the values of the type parameters. The values for each intrinsic type are described in 4.4. The values
 3 that objects of a derived type may assume are determined by the type definition, type parameter values,
 4 and the sets of values of its components.

5 2.4.3 Data entity

6 A **data entity** is a data object, the result of the evaluation of an expression, or the result of the execution
 7 of a function reference (called the function result). A data entity has a type and type parameters; it
 8 may have a data value (an exception is an undefined variable). Every data entity has a rank and is thus
 9 either a scalar or an array.

10 2.4.3.1 Data object

11 A **data object** (often abbreviated to **object**) is a constant (4.1.2), a variable (6), or a subobject of a
 12 constant. The type and type parameters of a named data object may be specified explicitly (5.2) or
 13 implicitly (5.5).

14 **Subobjects** are portions of certain objects that may be referenced and defined (variables only) inde-
 15 pendently of the other portions. These include portions of arrays (array elements and array sections),
 16 portions of character strings (substrings), portions of complex objects (real and imaginary parts), and
 17 portions of structures (components). Subobjects are themselves data objects, but subobjects are refer-
 18 enced only by object designators or intrinsic functions. A subobject of a variable is a variable. Subobjects
 19 are described in Clause 6.

20 Objects referenced by a name are:

21	a named scalar	(a scalar object)
	a named array	(an array object)

22 Subobjects referenced by an object designator are:

	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)
23	a substring	(a scalar subobject)

24 2.4.3.1.1 Variable

25 A **variable** may have a value and may be defined and redefined during execution of a program.

26 A named **local variable** of the scoping unit of a module, submodule, main program, or subprogram, is a
 27 named variable that is a local entity of the scoping unit, is not a dummy argument, is not in COMMON,
 28 does not have the BIND attribute, and is not accessed by use or host association. A named local variable
 29 of a BLOCK construct is a named variable that is declared in that construct, is not in COMMON, does
 30 not have the BIND attribute, and is not accessed by use association. A subobject of a named local
 31 variable is also a local variable.

32 2.4.3.1.2 Constant

33 A **constant** has a value and cannot become defined, redefined, or undefined during execution of a
 34 program. A constant with a name is called a **named constant** and has the PARAMETER attribute
 35 (5.3.12). A constant without a name is called a **literal constant** (4.4).

36 2.4.3.1.3 Subobject of a constant

- 1 A **subobject of a constant** is a portion of a constant. The portion referenced may depend on the
 2 value of a variable.

NOTE 2.6

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.

3 2.4.3.2 Expression

- 4 An **expression** (7.1) produces a data entity when evaluated. An expression represents either a data
 5 reference or a computation; it is formed from operands, operators, and parentheses. The type, type
 6 parameters, value, and rank of an expression result are determined by the rules in Clause 7.

7 2.4.3.3 Function reference

- 8 A **function reference** (12.5.3) produces a data entity when the function is executed during expression
 9 evaluation. The type, type parameters, and rank of a function result are determined by the interface of
 10 the function (12.3.3). The value of a function result is determined by execution of the function.

11 2.4.4 Scalar

- 12 A **scalar** (2.4.4) is a data entity that can be represented by a single value of the type and that is not an
 13 array (6.2). Its value, if defined, is a single element from the set of values that characterize its type.

NOTE 2.7

A structure is scalar even if it has arrays as components.

A scalar object of derived type has a single value that consists of the values of its components (4.5.8).

- 14 The **rank** of a scalar is zero. The shape of a scalar is represented by a rank-one array of size zero.

15 2.4.5 Array

- 16 An **array** is a set of scalar data, all of the same type and type parameters, whose individual elements
 17 are arranged in a rectangular pattern. An **array element** is one of the individual elements in the array
 18 and is a scalar. An **array section** is a subset of the elements of an array and is itself an array.

- 19 An array may have up to fifteen dimensions, and any **extent** (number of elements) in any dimension.
 20 The **rank** of the array is the number of dimensions; its **size** is the total number of elements, which is
 21 equal to the product of the extents. An array may have zero size. The **shape** of an array is determined
 22 by its rank and its extent in each dimension, and may be represented as a rank-one array whose elements
 23 are the extents. All named arrays shall be declared, and the rank of a named array is specified in its
 24 declaration. The rank of a named array, once declared, is constant; the extents may be constant or may
 25 vary during execution.

- 26 Two arrays are **conformable** if they have the same shape. A scalar is conformable with any array. Any

1 intrinsic operation defined for scalar objects may be applied to conformable objects. Such operations
 2 are performed element-by-element to produce a resultant array conformable with the array operands.
 3 Element-by-element operation means corresponding elements of the operand arrays are involved in a
 4 scalar operation to produce the corresponding element in the result array. Such an operation is described
 5 as **elemental**.

NOTE 2.8

If an elemental operation is intrinsically pure or is implemented by a pure elemental function (12.8), the element operations may be performed simultaneously or in any order.

6 A rank-one array may be constructed from scalars and other arrays and may be reshaped into any
 7 allowable array shape (4.7).

8 Arrays may be of any type and are described further in 6.2.

9 2.4.6 Co-array

10 A **co-array** is a data entity that has nonzero co-rank; it can be directly referenced or defined by any
 11 image. It may be a scalar or an array.

12 For each co-array on an image, there is a corresponding co-array with the same type, type parameters,
 13 and bounds on every other image. If a co-array is scalar, the set of corresponding co-arrays on all the
 14 images is arranged in a rectangular pattern. If a co-array is an array, the set of corresponding co-array
 15 elements on all the images is arranged in a rectangular pattern. In both cases, the dimensions of the
 16 pattern are called **co-dimensions**. The bounds for each co-dimension are called **co-bounds**.

17 A co-array on another image can be accessed directly by using *co-subscripts*. On its own image, a
 18 co-array can be accessed without use of co-subscripts.

19 The **co-rank** of a co-array is the number of co-dimensions. The **co-size** of a co-array is always equal to
 20 the number of images.

21 An object whose designator includes an *image-selector* is a **co-indexed object**. For a co-indexed object,
 22 its co-subscript list determines the image index in the same way that a subscript list determines the
 23 subscript order value for an array element (6.2.2.1).

24 Intrinsic procedures are provided for mapping between an image index and a list of co-subscripts.

NOTE 2.9

The mechanism for an image to reference and define a co-array on another image might vary according to the hardware. On a shared-memory machine, a co-array could be implemented as a section of an array of higher rank. On a distributed-memory machine with separate physical memory for each image, a processor might store a co-array at the same virtual address in each physical memory.

25 2.4.7 Pointer

26 A **data pointer** is a data entity that has the POINTER attribute. A **procedure pointer** is a procedure
 27 entity that has the POINTER attribute. A **pointer** is either a data pointer or a procedure pointer.

28 A pointer is **associated** with a **target** by pointer assignment (7.2.2) or pointer initialization (4.5.4.5,
 29 5.2.3). A data pointer may also be associated with a target by allocation (6.3.1) or argument association
 30 (12.5.2.8). A pointer is **disassociated** following execution of a NULLIFY statement, following pointer
 31 assignment with a disassociated pointer, by default initialization, or by explicit initialization. A data
 32 pointer may also be disassociated by execution of a DEALLOCATE statement. A disassociated pointer

- 1 is not associated with a target (16.5.2).
 2 A pointer that is not associated shall not be referenced or defined.
 3 If a data pointer is an array, the rank is declared, but the extents are determined when the pointer is
 4 associated with a target.

5 2.4.8 Storage

- 6 Many of the facilities of this part of ISO/IEC 1539 make no assumptions about the physical storage
 7 characteristics of data objects. However, program units that include storage association dependent
 8 features shall observe the storage restrictions described in 16.5.3.

9 2.5 Fundamental terms

- 10 For the purposes of this document, the terms and definitions in this subclause apply.

11 2.5.1 Name and designator

- 12 A **name** is used to identify a program constituent, such as a program unit, named variable, named
 13 constant, dummy argument, or derived type. The rules governing the construction of names are given
 14 in 3.2.1. A **designator** is a name followed by zero or more component selectors, complex part selectors,
 15 array section selectors, array element selectors, image selectors, and substring selectors.
 16 An **object designator** is a designator for a data object. A **procedure designator** is a designator for
 17 a procedure.

NOTE 2.10

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

18 2.5.2 Keyword

- 19 The term **keyword** is used in two ways.

- 20 (1) It is used to describe a word that is part of the syntax of a statement. These keywords are
 21 not reserved words; that is, names with the same spellings are allowed. In the syntax rules,
 22 such keywords appear literally. In descriptive text, this meaning is denoted by the term
 23 “keyword” without any modifier. Examples of statement keywords are IF, READ, UNIT,
 24 KIND, and INTEGER.
 25 (2) It is used to denote names that identify items in a list. In actual argument lists, type
 26 parameter lists, and structure constructors, items may be identified by a preceding *keyword*=
 27 rather than their position within the list. An **argument keyword** is the name of a dummy
 28 argument in the interface for the procedure being referenced, a **type parameter keyword**
 29 is the name of a type parameter in the type being specified, and a **component keyword**
 30 is the name of a component in a structure constructor.

- 31 R215 *keyword* **is** *name*

NOTE 2.11

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

2.5.3 Association

Association is name association (16.5.1), pointer association (16.5.2), storage association (16.5.3), or inheritance association (16.5.4). Name association is argument association, host association, use association, linkage association, or construct association.

Storage association causes different entities to use the same storage. Any association permits an entity to be identified by different names in the same scoping unit or by the same name or different names in different scoping units.

2.5.4 Declaration

The term **declaration** refers to the specification of attributes for various program entities. Often this involves specifying the type of a named data object or specifying the shape of a named array object.

2.5.5 Definition

The term **definition** is used in two ways.

- (1) It refers to the specification of derived types, enumerations, and procedures.
- (2) 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**. Similarly, when a pointer is associated with a target or nullified, its pointer association status is said to become **defined**. When the association status of a pointer is not predictable, its pointer association status is said to be **undefined**.

Clause 16 describes the ways in which variables become defined and undefined.

2.5.6 Reference

A **data object reference** is the appearance of the data object designator in a context requiring its value at that point during execution.

A **procedure reference** is the appearance of the procedure designator, operator symbol, or assignment symbol in a context requiring execution of the procedure at that point. An occurrence of user-defined derived-type input/output (10.7.6) or derived-type finalization (4.5.6.2) is also a procedure reference.

The appearance of a data object designator or procedure designator in an actual argument list does not constitute a reference to that data object or procedure unless such a reference is necessary to complete the specification of the actual argument.

A **module reference** is the appearance of a module name in a USE statement (11.2.2).

2.5.7 Intrinsic

The qualifier **intrinsic** has two meanings.

- (1) The qualifier signifies that the term to which it is applied is defined in this part of ISO/IEC 1539. Intrinsic applies to types, procedures, modules, assignment statements, and operators. All intrinsic types, procedures, assignments, and operators may be used in any scoping unit without further definition or specification. Intrinsic modules may be accessed by use association. Intrinsic procedures and modules defined in this part of ISO/IEC 1539 are called standard intrinsic procedures and standard intrinsic modules, respectively.

- 1 (2) The qualifier applies to procedures or modules that are provided by a processor but are not
2 defined in this part of ISO/IEC 1539 (13, 14, 15.2). Such procedures and modules are called
3 nonstandard intrinsic procedures and nonstandard intrinsic modules, respectively.

4 **2.5.8 Operator**

5 An **operator** specifies a computation involving one (unary operator) or two (binary operator) data
6 values (**operands**). This part of ISO/IEC 1539 specifies a number of intrinsic operators (e.g., the
7 arithmetic operators +, −, *, /, and ** with numeric operands and the logical operators .AND., .OR.,
8 etc. with logical operands). Additional operators may be defined within a program (4.5.5, 12.4.3.4).

9 **2.5.9 Sequence**

10 A **sequence** is a set ordered by a one-to-one correspondence with the numbers 1, 2, through *n*. The
11 number of elements in the sequence is *n*. A sequence may be empty, in which case it contains no elements.

12 **2.5.10 Companion processors**

13 A processor has one or more companion processors. A **companion processor** is a processor-dependent
14 mechanism by which global data and procedures may be referenced or defined. A companion processor
15 may be a mechanism that references and defines such entities by a means other than Fortran (12.6.3),
16 it may be the Fortran processor itself, or it may be another Fortran processor. If there is more than
17 one companion processor, the means by which the Fortran processor selects among them are processor
18 dependent.

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

NOTE 2.12

A companion processor might or might not be a mechanism that conforms to the requirements of the C International Standard.

For example, a processor may allow a procedure defined by some language other than Fortran or C to be invoked if it can be described by a C prototype as defined in 6.5.5.3 of the C International Standard.

3 Lexical tokens, source form, and macro processing

3.1 Processor character set

The processor character set is processor dependent. Each character in a processor character set is either a **control character** or a **graphic character**. The set of graphic characters is further divided into letters (3.1.1), digits (3.1.2), underscore (3.1.3), special characters (3.1.4), and other characters (3.1.5).

The letters, digits, underscore, and special characters make up the **Fortran character set**.

R301 *character* **is** *alphanumeric-character*
 or *special-character*

R302 *alphanumeric-character* **is** *letter*
 or *digit*
 or *underscore*

Except for the currency symbol, the graphics used for the characters shall be as given in 3.1.1, 3.1.2, 3.1.3, and 3.1.4. However, the style of any graphic is not specified.

3.1.1 Letters

The twenty-six **letters** are:

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

The set of letters defines the syntactic class *letter*. The processor character set shall include lower-case and upper-case letters. A lower-case letter is equivalent to the corresponding upper-case letter in program units except in a character context (3.3).

NOTE 3.1

The following statements are equivalent:

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

3.1.2 Digits

The ten **digits** are:

0 1 2 3 4 5 6 7 8 9

The ten digits define the syntactic class *digit*.

3.1.3 Underscore

R303 *underscore* **is** _

3.1.4 Special characters

- ¹ The **special characters** are shown in Table 3.1.

Table 3.1: **Special characters**

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

1 The special characters define the syntactic class *special-character*. Some of the special characters are
2 used for operator symbols, bracketing, and various forms of separating and delimiting other lexical
3 tokens.

4 **3.1.5 Other characters**

5 Additional characters may be representable in the processor, but may appear only in comments (3.3.1.1,
6 3.3.2.1), character constants (4.4.5), input/output records (9.1.1), and character string edit descriptors
7 (10.3.2).

8 **3.2 Low-level syntax**

9 The **low-level syntax** describes the fundamental lexical tokens of a program unit. **Lexical tokens** are
10 sequences of characters that constitute the building blocks of a program. They are keywords, names,
11 literal constants other than complex literal constants, operators, labels, delimiters, comma, =, =>, :, ::,
12 ;, and %.

13 **3.2.1 Names**

14 **Names** are used for various entities such as variables, program units, dummy arguments, named con-
15 stants, and derived types.

16 R304 *name* **is** *letter* [*alphanumeric-character*] ...

17 C301 (R304) The maximum length of a *name* is 63 characters.

NOTE 3.2

Examples of names:

```

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

```

NOTE 3.3

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

3.2.2 Constants

- 2 R305 *constant* **is** *literal-constant*
 3 **or** *named-constant*
- 4 R306 *literal-constant* **is** *int-literal-constant*
 5 **or** *real-literal-constant*
 6 **or** *complex-literal-constant*
 7 **or** *logical-literal-constant*
 8 **or** *char-literal-constant*
 9 **or** *bits-literal-constant*
- 10 R307 *named-constant* **is** *name*
- 11 R308 *int-constant* **is** *constant*
- 12 C302 (R308) *int-constant* shall be of type integer.
- 13 R309 *char-constant* **is** *constant*
- 14 C303 (R309) *char-constant* shall be of type character.

3.2.3 Operators

- 16 R310 *intrinsic-operator* **is** *power-op*
 17 **or** *mult-op*
 18 **or** *add-op*
 19 **or** *concat-op*
 20 **or** *rel-op*
 21 **or** *not-op*
 22 **or** *and-op*
 23 **or** *or-op*
 24 **or** *equiv-op*
- 25 R707 *power-op* **is** ******
- 26 R708 *mult-op* **is** *****
 27 **or** **/**
- 28 R709 *add-op* **is** **+**
 29 **or** **–**
- 30 R711 *concat-op* **is** **//**
- 31 R713 *rel-op* **is** **.EQ.**
 32 **or** **.NE.**
 33 **or** **.LT.**
 34 **or** **.LE.**
 35 **or** **.GT.**
 36 **or** **.GE.**
 37 **or** **==**

1		or /=
2		or <
3		or <=
4		or >
5		or >=
6	R718 <i>not-op</i>	is .NOT.
7	R719 <i>and-op</i>	is .AND.
8	R720 <i>or-op</i>	is .OR.
9	R721 <i>equiv-op</i>	is .EQV.
10		or .NEQV.
11		or .XOR.
12	R311 <i>defined-operator</i>	is <i>defined-unary-op</i>
13		or <i>defined-binary-op</i>
14		or <i>extended-intrinsic-op</i>
15	R703 <i>defined-unary-op</i>	is . <i>letter</i> [<i>letter</i>]
16	R723 <i>defined-binary-op</i>	is . <i>letter</i> [<i>letter</i>]
17	R312 <i>extended-intrinsic-op</i>	is <i>intrinsic-operator</i>

18 3.2.4 Statement labels

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

20 R313 *label* **is** *digit* [*digit* [*digit* [*digit* [*digit*]]]]

21 C304 (R313) At least one digit in a *label* shall be nonzero.

22 If a statement is labeled, the statement shall contain a nonblank character. The same statement label
 23 shall not be given to more than one statement in a scoping unit. Leading zeros are not significant in
 24 distinguishing between statement labels.

NOTE 3.4

For example:

99999

10

010

are all statement labels. The last two are equivalent.

There are 99999 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.

25 Any statement may have a statement label, but the labels are used only in the following ways.

- 26 (1) The label on a branch target statement (8.2) is used to identify that statement as the
 27 possible destination of a branch.
- 28 (2) The label on a FORMAT statement (10.2.1) is used to identify that statement as the format

- 1 specification for a data transfer statement (9.5).
 2 (3) In some forms of the DO construct (8.1.7), the range of the DO construct is identified by
 3 the label on the last statement in that range.

4 3.2.5 Delimiters

5 **Delimiters** are used to enclose syntactic lists. The following pairs are delimiters:

- 6 (...)
 7 / ... /
 8 [...]
 9 (/ ... /)

10 3.3 Source form

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

15 A **character context** means characters within a character literal constant (4.4.5) or within a character
 16 string edit descriptor (10.3.2).

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

18 There are two **source forms**: free and fixed. Free form and fixed form shall not be mixed in the same program unit.

19 The means for specifying the source form of a program unit are processor dependent.

20 3.3.1 Free source form

21 In **free source form** there are no restrictions on where a statement (or portion of a statement) may
 22 appear within a line. A line may contain zero characters. If a line consists entirely of characters of
 23 default kind (4.4.5), it may contain at most 132 characters. If a line contains any character that is not
 24 of default kind, the maximum number of characters allowed on the line is processor dependent.

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

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

NOTE 3.5

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

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

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

Adjacent keywords where separating blanks are optional

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

1 3.3.1.1 Free form commentary

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

NOTE 3.6

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

8 3.3.1.2 Free form statement continuation

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

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

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

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

25 3.3.1.3 Free form statement termination

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

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

5 **3.3.1.4 Free form statements**

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

NOTE 3.7

No Fortran statement begins with a digit.

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

8 **3.3.2 Fixed source form**

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

12 Except in a character context, blanks are insignificant and may be used freely throughout the program.

13 **3.3.2.1 Fixed form commentary**

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

NOTE 3.8

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

20 **3.3.2.2 Fixed form statement continuation**

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

NOTE 3.9

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

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

26 **3.3.2.3 Fixed form statement termination**

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

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

1 3.3.2.4 Fixed form statements

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

7 3.4 Including source text

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

10 `INCLUDE char-literal-constant`

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

12 An INCLUDE line is not a Fortran statement.

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

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

22 When an INCLUDE line is resolved, the first included statement line shall not be a continuation line
23 and the last included statement line shall not be continued.

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

NOTE 3.10

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

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

26 3.5 Macro processing

1 3.5.1 Macro definition

2 A macro definition defines a macro. A defined macro shall only be referenced by a USE statement,
3 IMPORT statement, or macro expansion statement. A defined macro shall not be redefined.

```

4 R314    macro-definition           is   define-macro-stmt
5                                           [ macro-declaration-stmt ] ...
6                                           macro-body-block
7                                           end-macro-stmt

```

8	R315	<i>define-macro-stmt</i>	is	DEFINE MACRO [, <i>macro-attr-list</i>] :: <i>macro-name</i> ■ ■ [([<i>macro-dummy-arg-name-list</i>])]
9				

10 C305 (R315) A *macro-dummy-arg-name* shall not appear more than once in a *macro-dummy-arg-*
11 *name-list*.

12 R316 *macro-attr* is *access-spec*

13 The DEFINE MACRO statement begins the definition of the macro *macro-name*. Appearance of an
14 *access-spec* in the DEFINE MACRO statement explicitly gives the macro the specified attribute. Each
15 *macro-dummy-arg-name* is a macro dummy argument. A macro dummy argument is a macro local
16 variable.

17	R317	<i>macro-declaration-stmt</i>	is	<i>macro-type-declaration-stmt</i>
18			or	<i>macro-optional-decl-stmt</i>
19			or	<i>macro-variable-decl-stmt</i>

20 R318 *macro-type-declaration-stmt* **is** MACRO *macro-type-spec* :: *macro-local-variable-name-list*

21 R319 *macro-optional-decl-stmt* is MACRO OPTIONAL :: *macro-dummy-arg-name-list*

```
22  R320  macro-variable-decl-stmt    is  MACRO VARIABLE :: macro-local-variable-name-list
```

```

23  R321  macro-type-spec      is  INTEGER [ ( [ KIND= ] macro-expr ) ]

```

24 C306 (R318, *macro-variable-decl-stmt*) A *macro-local-variable-name* shall not be the same as the name
25 of a dummy argument of the macro being defined.

26 C307 (R319) A *macro-dummy-arg-name* shall be the name of a dummy argument of the macro being
27 defined.

28 C308 (R321) If *macro-expr* appears, when the macro is expanded *macro-expr* shall be of type integer,
29 and have a non-negative value that specifies a representation method that exists on the processor.

30 A macro type declaration statement specifies that the named entities are macro local variables of the
31 specified type. If the kind is not specified, they are of default kind. A macro variable declaration
32 statement declares untyped macro local variables; the value of an untyped macro local variable is a
33 token sequence, and its initial value is an empty sequence (no tokens). A macro local variable that is
34 not a macro dummy argument shall appear in a macro type declaration statement or in a macro variable
35 declaration statement.

36 R322 *macro-body-block* is [*macro-body-construct*] ...

```

37 R323    macro-body-construct      is macro-definition
38                                     or expand-stmt
39                                     or macro-body-stmt
40                                     or macro-do-construct
41                                     or macro-if-construct

```

- 1 **or** *macro-int-assignment-stmt*
2 **or** *macro-tok-assignment-stmt*
- 3 C309 A statement in a macro definition that is not a *macro-body-construct* or *macro-definition* shall
4 not appear on a line with any other statement.
- 5 R324 *macro-do-construct* **is** *macro-do-stmt*
6 *macro-body-block*
7 *macro-end-do-stmt*
- 8 R325 *macro-do-stmt* **is** MACRO DO *macro-do-variable-name* = *macro-do-limit* , ■
9 ■ *macro-do-limit* [, *macro-do-limit*]
- 10 C310 (R325) A *macro-do-variable-name* shall be a local variable of the macro being defined, and shall
11 not be a macro dummy argument.
- 12 R326 *macro-do-limit* **is** *macro-expr*
- 13 C311 (R326) A *macro-do-limit* shall expand to an expression of type integer.
- 14 R327 *macro-end-do-stmt* **is** MACRO END DO
- 15 A macro DO construct iterates the expansion of its enclosed macro body block at macro expansion time.
16 The number of iterations is determined by the values of the expanded macro expressions in the MACRO
17 DO statement.
- 18 R328 *macro-if-construct* **is** *macro-if-then-stmt*
19 *macro-body-block*
20 [*macro-else-if-stmt*
21 *macro-body-block*] ...
22 [*macro-else-stmt*
23 *macro-body-block*]
24 *macro-end-if-stmt*
- 25 R329 *macro-if-then-stmt* **is** MACRO IF (*macro-condition*) THEN
- 26 R330 *macro-else-if-stmt* **is** MACRO ELSE IF (*macro-condition*) THEN
- 27 R331 *macro-else-stmt* **is** MACRO ELSE
- 28 R332 *macro-end-if-stmt* **is** MACRO END IF
- 29 R333 *macro-condition* **is** *macro-expr*
- 30 C312 (R333) A macro condition shall expand to an expression of type logical.
- 31 A macro IF construct provides conditional expansion of its enclosed macro body blocks at macro expansion
32 time. Whether the enclosed macro body blocks contribute to the macro expansion is determined by
33 the logical value of the expanded macro expressions in the MACRO IF and MACRO ELSE IF statements.
- 34 R334 *macro-int-assignment-stmt* **is** MACRO *macro-integer-variable-name* = *macro-expr*
- 35 C313 (R334) *macro-integer-variable-name* shall be the name of a macro local variable of type integer.
- 36 R335 *macro-tok-assignment-stmt* **is** MACRO *macro-tok-variable-name* = *assignment-tok-sequence*
37
- 38 C314 (R335) *macro-tok-variable-name* shall be the name of an untyped macro local variable that is
39 not a macro dummy argument.

- 1 R336 *assignment-tok-sequence* is *result-token* [*result-token*] ... [&&]
- 2 R337 *macro-body-stmt* is *result-token* [*result-token*] ... [&&]
- 3 C315 (R337) If the first *result-token* is MACRO the second *result-token* shall not be a keyword or
4 name.
- 5 C316 (R337) If the first token is DEFINE the second token shall not be MACRO.
- 6 R338 *result-token* is *token* [%% *token*] ...
- 7 R339 *token* is any lexical token including labels, keywords, and semi-colon.
- 8 C317 && shall not appear in the last *macro-body-stmt* of a macro definition.
- 9 C318 When a macro is expanded, the last *macro-body-stmt* processed shall not end with &&.
- 10 R340 *end-macro-stmt* is END MACRO [*macro-name*]
- 11 C319 (R314) The *macro-name* in the END MACRO statement shall be the same as the *macro-name*
12 in the DEFINE MACRO statement.
- 13 R341 *macro-expr* is *basic-token-sequence*
- 14 C320 (R341) A *macro-expr* shall expand to a scalar initialization expression.
- 15 Macro expressions are used to control the behavior of the MACRO DO and MACRO IF constructs when
16 a macro is being expanded. The type, type parameters, and value of a macro expression are determined
17 when that macro expression is expanded.

18 3.5.2 Macro expansion

19 3.5.2.1 General

20 Macro expansion is the conceptual replacement of the EXPAND statement with the Fortran statements
21 that it produces. The semantics of an EXPAND statement are those of the Fortran statements that it
22 produces. It is recommended that a processor be capable of displaying the results of macro expansion. It
23 is processor-dependent whether comments in a macro definition appear in the expansion. It is processor-
24 dependent whether continuations and consecutive blanks that are not part of a token are preserved.

25 The process of macro expansion produces Fortran statements consisting of tokens. The combined length
26 of the tokens for a single statement, plus inter-token spacing, shall not be greater than 33280 characters.
27 If a statement contains any character that is not of default kind, the maximum number of characters
28 allowed is processor dependent.

NOTE 3.11

This length is so that the result of macro expansion can be formed into valid free form Fortran source, consisting of an initial line and 255 continuation lines, times 130 which allows for beginning and ending continuation characters (&) on each line.

Also, breaking tokens across continuation lines in macro definitions and in EXPAND statements does not affect macro expansion: it is as if they were joined together before replacement.

- 29 R342 *expand-stmt* is EXPAND *macro-name* [(*macro-actual-arg-list*)]
- 30 C321 (R342) *macro-name* shall be the name of a macro that was previously defined or accessed via
31 use or host association.

- 1 C322 (R342) The macro shall expand to a sequence or zero or more complete Fortran statements.
- 2 C323 (R342) The statements produced by a macro expansion shall conform to the syntax rules and
3 constraints as if they replaced the EXPAND statement prior to program processing.
- 4 C324 (R342) The statements produced by a macro expansion shall not include a statement which
5 ends the scoping unit containing the EXPAND statement.
- 6 C325 (R342) If a macro expansion produces a statement which begins a new scoping unit, it shall also
7 produce a statement which ends that scoping unit.
- 8 C326 (R342) If the EXPAND statement appears as the *action-stmt* of an *if-stmt*, it shall expand
9 to exactly one *action-stmt* that is not an *end-function-stmt*, *end-mp-subprogram-stmt*, *end-*
10 *program-stmt*, *end-subroutine-stmt*, or *if-stmt*.
- 11 C327 (R342) If the EXPAND statement appears as a *do-term-action-stmt*, it shall expand to exactly one *action-*
12 *stmt* that is not an *arithmetic-if-stmt*, *continue-stmt*, *cycle-stmt*, *end-function-stmt*, *end-mp-subprogram-stmt*,
13 *end-program-stmt*, *end-subroutine-stmt*, *exit-stmt*, *goto-stmt*, *return-stmt*, or *stop-stmt*.
- 14 C328 (R342) If the EXPAND statement has a label, the expansion of the macro shall produce at least
15 one statement, and the first statement produced shall not have a label.
- 16 C329 (R342) A *macro-actual-arg* shall appear corresponding to each nonoptional macro dummy ar-
17 gument.
- 18 C330 (R342) At most one *macro-actual-arg* shall appear corresponding to each optional macro dummy
19 argument.
- 20 Expansion of a macro is performed by the EXPAND statement. If the EXPAND statement has a label,
21 the label is interpreted after expansion as belonging to the first statement of the expansion.
- 22 R343 *macro-actual-arg* **is** [*macro-dummy-name* =] *macro-actual-arg-value*
- 23 C331 (R343) *macro-dummy-name* shall be the name of a macro dummy argument of the macro being
24 expanded.
- 25 C332 (R342) The *macro-dummy-name*= shall not be omitted unless it has been omitted from each
26 preceding *macro-actual-arg* in the *expand-stmt*.
- 27 R344 *macro-actual-arg-value* **is** *basic-token-sequence*
- 28 R345 *basic-token-sequence* **is** *basic-token*
29 **or** [*basic-token-sequence*] *nested-token-sequence* ■
30 ■ [*basic-token-sequence*]
31 **or** *basic-token basic-token-sequence*
- 32 R346 *basic-token* **is** any lexical token except comma, parentheses, array ■
33 ■ constructor delimiters, and semi-colon.
- 34 R347 *nested-token-sequence* **is** ([*arg-token*] ...)
35 **or** (/ [*arg-token*] ... /)
36 **or** *lbracket* [*arg-token*] ... *rbracket*
- 37 R348 *arg-token* **is** *basic-token*
38 **or** ,

39 Macro expansion processes any macro declarations of the macro definition, and then expands its macro
40 body block. Any macro expressions in *macro-type-specs* are evaluated and the kinds of the macro

- 1 variables thereby declared are determined for that particular expansion.
- 2 Macro expansion of a macro body block processes each macro body construct of the macro body block
3 in turn, starting with the first macro body construct and ending with the last macro body construct.
- 4 Expansion of a statement within a macro body construct consists of three steps:
- 5 (1) token replacement,
 - 6 (2) token concatenation, and
 - 7 (3) statement-dependent processing.

8 3.5.2.2 Token replacement

- 9 Token replacement replaces each token of a macro body statement, assignment token sequence, or macro
10 expression that is a macro local variable with the value of that variable.
- 11 In a macro expression, a reference to the PRESENT intrinsic function with a macro dummy argument
12 name as its actual argument is replaced by the token `.TRUE.` if the specified macro dummy argument
13 is present, and the token `.FALSE.` if the specified macro dummy argument is not present. Otherwise,
14 the value of a macro dummy argument that is present is the sequence of tokens from the corresponding
15 actual argument, and the value of a macro dummy argument that is not present is a zero-length token
16 sequence.
- 17 The value of an integer macro variable is its minimal-length decimal representation; if negative this
18 produces two tokens, a minus sign and an unsigned integer literal constant. An untyped macro local
19 variable expands to the sequence of tokens that was assigned to it, or to zero-length token sequence if it
20 has never been assigned to.

21 3.5.2.3 Token concatenation

- 22 Token concatenation is performed with the `%%` operator, which is only permitted inside a macro defini-
23 tion. After expansion, each sequence of single tokens separated by `%%` operators is replaced by a single
24 token consisting of the concatenated text of the sequence of tokens. The result of a concatenation shall
25 be a valid Fortran token, and may be a different kind of token from one or more of the original sequence
26 of tokens.
- 27 C333 (R338) The result of token concatenation shall have the form of a lexical token.

NOTE 3.12

For example, the sequence

```
3 %% .14159 %% E %% + %% 0
```

forms the single real literal constant 3.14159E+0.

28 3.5.2.4 Macro body statements

- 29 Processing a macro body statement produces a whole or partial Fortran statement. A macro body
30 statement that is either the first macro body statement processed by this macro expansion or the next
31 macro body statement processed after a macro body statement that did not end with the continuation
32 generation operator `&&`, is an initial macro body statement. The next macro body statement processed
33 after a macro body statement that ends with `&&` is a continuation macro body statement. An initial
34 macro body statement that does not end with `&&` produces a whole Fortran statement consisting of its
35 token sequence. All other macro body statements produce partial Fortran statements, and the sequence
36 of tokens starting with those produced by the initial macro body statement and appending the tokens

1 produced by each subsequent continuation macro body statement form a Fortran statement. The &&
2 operators are not included in the token sequence.

3 3.5.2.5 The MACRO DO construct

4 The MACRO DO construct specifies the repeated expansion of a macro body block. Processing the
5 MACRO DO statement performs the following steps in sequence.

- 6 (1) The initial parameter m_1 , the terminal parameter m_2 , and the incrementation parameter
7 m_3 are of type integer with the same kind type parameter as the *macro-do-variable-name*.
8 Their values are given by the first *macro-expr*, the second *macro-expr*, and the third *macro-*
9 *expr* of the *macro-do-stmt* respectively, including, if necessary, conversion to the kind type
10 parameter of the *macro-do-variable-name* according to the rules for numeric conversion
11 (Table 7.13). If the third *macro-expr* does not appear, m_3 has the value 1. The value of m_3
12 shall not be zero.
- 13 (2) The MACRO DO variable becomes defined with the value of the initial parameter m_1 .
- 14 (3) The **iteration count** is established and is the value of the expression $(m_2 - m_1 + m_3)/m_3$,
15 unless that value is negative, in which case the iteration count is 0.

16 After this, the following steps are performed repeatedly until processing of the MACRO DO construct
17 is finished.

- 18 (1) The iteration count is tested. If it is zero, the loop terminates and processing of the MACRO
19 DO construct is finished.
- 20 (2) If the iteration count is nonzero, the macro body block of the MACRO DO construct is
21 expanded.
- 22 (3) The iteration count is decremented by one. The MACRO DO variable is incremented by
23 the value of the incrementation parameter m_3 .

24 3.5.2.6 The MACRO IF construct

25 The MACRO IF construct provides conditional expansion of macro body blocks. At most one of the
26 macro body blocks of the MACRO IF construct is expanded. The macro conditions of the construct
27 are evaluated in order until a true value is found or a MACRO ELSE or MACRO END IF statement is
28 encountered. If a true value or a MACRO ELSE statement is found, the macro body block immediately
29 following is expanded and this completes the processing of the construct. If none of the evaluated
30 conditions is true and there is no MACRO ELSE statement, the processing of the construct is completed
31 without expanding any of the macro body blocks within the construct.

32 3.5.2.7 Macro assignment

33 Processing a macro integer assignment statement sets the macro local variable value to that of the macro
34 expression.

35 Processing a macro token assignment statement sets the macro local variable value to be the sequence
36 of tokens following the equals sign. If no tokens appear after the equals sign, the macro local variable is
37 set to the zero-length token sequence.

38 3.5.2.8 Macro definitions

39 Processing a macro definition defines a new macro. If a macro definition is produced by a macro expansion,
40 all of the statements of the produced macro definition have token replacement and concatenation
41 applied to them before the new macro is defined.

1 3.5.2.9 Examples

NOTE 3.13

This is a macro which loops over an array of any rank and processes each array element.

```

DEFINE MACRO loop_over(array,rank,traceinfo)
  MACRO INTEGER :: i
    BLOCK
      MACRO DO i=1,rank
        INTEGER loop_over_temp_%%i
      MACRO END DO
      MACRO DO i=1,rank
        DO loop_over_temp_%%i=1,size(array,i)
      MACRO END DO
        CALL impure_scalar_procedure(array(loop_over_temp_%%i &&
      MACRO DO i=2,rank
        ,loop_over_temp%i &&
      MACRO END DO
        ),traceinfo)
      MACRO DO i=1,rank
        END DO
      MACRO END DO
    END BLOCK
  END MACRO

```

NOTE 3.14

One can effectively pass macro names as macro arguments, since expansion of arguments occurs before analysis of each macro body statement. For example:

```

DEFINE MACRO :: iterator(count,operation)
  MACRO DO i=1,count
    EXPAND operation(i)
  MACRO END DO
END MACRO

DEFINE MACRO :: process_element(j)
  READ *,a(j)
  result(j) = process(a(j))
  IF (j>1) PRINT *,'difference =',result(j)-result(j-1)
END MACRO

EXPAND iterator(17,process_element)

```

This expands into 17 sets of 3 statements:

```

  READ *,a(1)
  result(1) = process(a(1))
  IF (1>1) PRINT *,'difference =',result(1)-result(1-1)
  READ *,a(2)
  result(2) = process(a(2))
  IF (2>1) PRINT *,'difference =',result(2)-result(2-1)
  ...

```

NOTE 3.14 (cont.)

```

READ *,a(17)
result(17) = process(a(17))
IF (17>1) PRINT *,'difference =',result(17)-result(17-1)

```

NOTE 3.15

Using the ability to evaluate initialization expressions under macro control and test them, one can create interfaces and procedures for all kinds of a type, for example:

```

DEFINE MACRO :: i_square_procs()
  MACRO INTEGER i
  MACRO DO i=1,1000
    MACRO IF (SELECTED_INT_KIND(i)>=0 .AND.
              (i==1 .OR. SELECTED_INT_KIND(i)/=SELECTED_INT_KIND(i-1))) THEN
      FUNCTION i_square_range_%%i(a) RESULT(r)
        INTEGER(SELECTED_INT_KIND(i)) a,r
        r = a**2
      END FUNCTION
    MACRO END IF
  MACRO END DO
END MACRO

```

NOTE 3.16

Macros can be used to define other macros on expansion. For example,

```

! Macro that defines a macro which assigns a value to an array element
DEFINE MACRO :: assign_shortcut(rank)
  DEFINE MACRO assign_%%rank(array,indices,value)
    MACRO INTEGER :: i
    array(indices(1)&&
    MACRO DO i=2,rank
      ,indices(i)&&
    MACRO END DO
      )=value
  END MACRO assign_%%rank
END MACRO assign_shortcut

! Create assignment macros for all ranks
MACRO DO i=1,15
  EXPAND assign_shortcut(i)
MACRO END DO

! Now use the rank-3 assignment macro:
REAL :: A(10,10,10)
INTEGER :: indices(3)=[1,5,6]
EXPAND assign_3(A,indices,5.0)
! Expands to:
! A(indices(1),indices(2),indices(3))=5.0

```

4 Types

4.1 The concept of type

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.

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.

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

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

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

4.1.1 Set of values

For each type, there is a set of valid values. The sets of valid values for logical and bits are completely determined by this part of ISO/IEC 1539. The sets of valid values for integer, character, and real are processor dependent. The set of valid values for complex consists of the set of all the combinations of the values of the individual components. The set of valid values for a derived type is as defined in 4.5.8.

4.1.2 Constants

The syntax for literal constants of each intrinsic type is specified in 4.4.

The syntax for denoting a value indicates the type, type parameters, and the particular value.

A constant value can be given a name (5.3.12, 5.4.10).

A structure constructor (4.5.10) can be used to construct a constant value of derived type from an appropriate sequence of initialization expressions (7.1.12). Such a constant value is scalar even if it has components that are arrays.

4.1.3 Operations

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

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

4.2 Type parameters

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

- 1 The intrinsic types are all parameterized. Derived types may be defined to be parameterized.
- 2 A type parameter is either a kind type parameter or a length type parameter. All type parameters are
3 of type integer.
- 4 A kind type parameter may be used in initialization and specification expressions within the derived-type
5 definition (4.5.2) for the type; it participates in generic resolution (12.5.5.2). Each of the intrinsic types
6 has a kind type parameter named KIND, which is used to distinguish multiple representations of the
7 intrinsic type.

NOTE 4.1

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

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

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

NOTE 4.2

The adjective “length” is used for type parameters other than kind type parameters because they often specify a length, as for intrinsic character type. However, they may 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.

- 11 A type parameter value may be specified with a type specification (4.4, 4.5.9).
- 12 R401 *type-param-value* **is** *scalar-int-expr*
13 **or** *
- 14 **or** :
- 15 C401 (R401) The *type-param-value* for a kind type parameter shall be an initialization expression.
- 16 C402 (R401) A colon shall not be used as a *type-param-value* except in the declaration of an entity or
17 component that has the POINTER or ALLOCATABLE attribute.
- 18 A **deferred type parameter** is a length type parameter whose value can change during execution of
19 the program. A colon as a *type-param-value* specifies a deferred type parameter.
- 20 The values of the deferred type parameters of an object are determined by successful execution of an
21 ALLOCATE statement (6.3.1), execution of an intrinsic assignment statement (7.2.1.3), execution of a
22 pointer assignment statement (7.2.2), or by argument association (12.5.2).

NOTE 4.3

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

1 An **assumed type parameter** is a length type parameter for a dummy argument that assumes the
 2 type parameter value from the corresponding actual argument; it is also used for an associate name in a
 3 SELECT TYPE construct that assumes the type parameter value from the corresponding selector, and
 4 for a named constant of type character that assumes its length from the *initialization-expr*. An asterisk
 5 as a *type-param-value* specifies an assumed type parameter.

6 4.3 Relationship of types and values to objects

7 The name of a type serves as a type specifier and may be used to declare objects of that type. A
 8 declaration specifies the type of a named object. A data object may be declared explicitly or implicitly.
 9 Data objects may have attributes in addition to their types. Clause 5 describes the way in which a data
 10 object is declared and how its type and other attributes are specified.

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

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

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

18 4.3.1 Type specifiers and type compatibility

19 4.3.1.1 General

20 A type is specified by a **type specifier**.

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

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

24 R403 *declaration-type-spec* **is** *intrinsic-type-spec*
 25 **or** TYPE (*intrinsic-type-spec*)
 26 **or** TYPE (*derived-type-spec*)
 27 **or** CLASS (*derived-type-spec*)
 28 **or** CLASS (*)

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

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

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

34 C407 An entity declared with the CLASS keyword shall be a dummy argument or have the ALLO-
 35 CATABLE or POINTER attribute. It shall not have the VALUE attribute.

36 An *intrinsic-type-spec* specifies the named intrinsic type and its type parameter values. A *derived-type-*
 37 *spec* specifies the named derived type and its type parameter values.

NOTE 4.4

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

1 **4.3.1.2 TYPE**

2 A TYPE type specifier is used to declare entities of an intrinsic or derived type.

3 Where a data entity is declared explicitly using the TYPE type specifier to be of derived type, the
 4 specified derived type shall have been defined previously in the scoping unit or be accessible there by
 5 use or host association. If the data entity is a function result, the derived type may be specified in
 6 the FUNCTION statement provided the derived type is defined within the body of the function or is
 7 accessible there by use or host association. If the derived type is specified in the FUNCTION statement
 8 and is defined within the body of the function, it is as if the function result variable was declared with
 9 that derived type immediately following the *derived-type-def* of the specified derived type.

10 **4.3.1.3 CLASS**

11 A **polymorphic** entity is a data entity that is able to be of differing types during program execution.
 12 The type of a data entity at a particular point during execution of a program is its **dynamic type**. The
 13 **declared type** of a data entity is the type that it is declared to have, either explicitly or implicitly.

14 A CLASS type specifier is used to declare polymorphic entities. The declared type of a polymorphic
 15 entity is the specified type if the CLASS type specifier contains a type name.

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

19 A nonpolymorphic entity is **type compatible** only with entities of the same declared type. A poly-
 20 morphic entity that is not an unlimited polymorphic entity is type compatible with entities of the same
 21 declared type or any of its extensions. Even though an unlimited polymorphic entity is not considered
 22 to have a declared type, it is type compatible with all entities. An entity is type compatible with a type
 23 if it is type compatible with entities of that type.

24 Two entities are **type incompatible** if neither is type compatible with the other.

NOTE 4.5

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.

25 A polymorphic allocatable object may be allocated to be of any type with which it is type compatible.
 26 A polymorphic pointer or dummy argument may, during program execution, be associated with objects
 27 with which it is type compatible.

28 The dynamic type of an allocated allocatable polymorphic object is the type with which it was allocated.

1 The dynamic type of an associated polymorphic pointer is the dynamic type of its target. The dynamic
 2 type of a nonallocatable nonpointer polymorphic dummy argument is the dynamic type of its associated
 3 actual argument. The dynamic type of an unallocated allocatable or a disassociated pointer is the same
 4 as its declared type. The dynamic type of an entity identified by an associate name (8.1.3) is the dynamic
 5 type of the selector with which it is associated. The dynamic type of an object that is not polymorphic
 6 is its declared type.

NOTE 4.6

Only components of the declared type of a polymorphic object may be designated by component selection (6.1.2).

7 4.4 Intrinsic types

8 4.4.1 Classification and specification

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

11 The numeric types are provided for numerical computation. The normal operations of arithmetic,
 12 addition (+), subtraction (−), multiplication (*), division (/), exponentiation (**), identity (unary +),
 13 and negation (unary −), are defined intrinsically for the numeric types.

14 R404	<i>intrinsic-type-spec</i>	is INTEGER [<i>kind-selector</i>]
15		or REAL [<i>kind-selector</i>]
16		or DOUBLE PRECISION
17		or COMPLEX [<i>kind-selector</i>]
18		or CHARACTER [<i>char-selector</i>]
19		or LOGICAL [<i>kind-selector</i>]
20		or BITS [<i>kind-selector</i>]

21 R405	<i>kind-selector</i>	is ([KIND =] <i>scalar-int-initialization-expr</i>)

22 C408	(R405) The value of <i>scalar-int-initialization-expr</i> shall be nonnegative and shall specify a rep-
23	resentation method that exists on the processor.

24 4.4.2 Integer type

25 The set of values for the **integer type** is a subset of the mathematical integers. The processor shall
 26 provide one or more **representation methods** that define sets of values for data of type integer. Each
 27 such method is characterized by a value for a type parameter called the **kind** type parameter; this kind
 28 type parameter is of type default integer. The kind type parameter of a representation method is returned
 29 by the intrinsic inquiry function KIND (13.7.96). The decimal exponent range of a representation method
 30 is returned by the intrinsic function RANGE (13.7.143). The intrinsic function SELECTED_INT_KIND
 31 (13.7.153) returns a kind value based on a specified decimal range requirement. The integer type includes
 32 a zero value, which is considered to be neither negative nor positive. The value of a signed integer zero
 33 is the same as the value of an unsigned integer zero.

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

36 The type specifier for the integer type uses the keyword INTEGER.

37 If the kind type parameter is not specified, the default kind value is KIND (0) and the type specified is
 38 **default integer**. The decimal exponent range of default integer shall be at least 5.

- 1 Any integer value may be represented as a *signed-int-literal-constant*.
- 2 R406 *signed-int-literal-constant* **is** [*sign*] *int-literal-constant*
- 3 R407 *int-literal-constant* **is** *digit-string* [_ *kind-param*]
- 4 R408 *kind-param* **is** *digit-string*
- 5 **or** *scalar-int-constant-name*
- 6 R409 *signed-digit-string* **is** [*sign*] *digit-string*
- 7 R410 *digit-string* **is** *digit* [*digit*] ...
- 8 R411 *sign* **is** +
- 9 **or** -
- 10 C409 (R408) A *scalar-int-constant-name* shall be a named constant of type integer.
- 11 C410 (R408) The value of *kind-param* shall be nonnegative.
- 12 C411 (R407) The value of *kind-param* shall specify a representation method that exists on the pro-
- 13 cessor.
- 14 The optional kind type parameter following *digit-string* specifies the kind type parameter of the integer
- 15 constant; if it is not present, the constant is of type default integer.
- 16 An integer constant is interpreted as a decimal value.

NOTE 4.7

Examples of signed integer literal constants are:

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

where SHORT is a scalar integer named constant.

17 4.4.3 Real type

- 18 The **real type** has values that approximate the mathematical real numbers. The processor shall provide
- 19 two or more **approximation methods** that define sets of values for data of type real. Each such method
- 20 has a **representation method** and is characterized by a value for a type parameter called the **kind**
- 21 type parameter; this kind type parameter is of type default integer. The kind type parameter of an
- 22 approximation method is returned by the intrinsic inquiry function KIND (13.7.96).
- 23 The decimal precision, decimal exponent range, and radix of an approximation method are returned by
- 24 the intrinsic functions PRECISION (13.7.137), RANGE (13.7.143), and RADIX (13.7.140). The intrinsic
- 25 function SELECTED_REAL_KIND (13.7.154) returns a kind value based on specified precision, range,
- 26 and radix requirements.

NOTE 4.8

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

1 The real type includes a zero value. Processors that distinguish between positive and negative zeros
2 shall treat them as mathematically equivalent

- 3 (1) in all relational operations,
- 4 (2) as actual arguments to intrinsic procedures other than those for which it is explicitly specified
5 that negative zero is distinguished, and
- 6 (3) as the *scalar-numeric-expr* in an arithmetic IF.

NOTE 4.9

On a processor that can distinguish between 0.0 and -0.0 ,

(X >= 0.0)

evaluates to true if $X = 0.0$ or if $X = -0.0$,

(X < 0.0)

evaluates to false for $X = -0.0$, and

IF (X) 1,2,3

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

In order to distinguish between 0.0 and -0.0 , a program should use the SIGN function. 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 .

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

9 If the type keyword REAL is specified and the kind type parameter is not specified, the default kind
10 value is KIND (0.0) and the type specified is **default real**. If the type keyword DOUBLE PRECISION
11 is specified, the kind value is KIND (0.0D0) and the type specified is **double precision real**. The
12 decimal precision of the double precision real approximation method shall be greater than that of the
13 default real method.

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

17 R412 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*

18 R413 *real-literal-constant* **is** *significand* [*exponent-letter exponent*] [*_ kind-param*]
19 **or** *digit-string exponent-letter exponent* [*_ kind-param*]

20 R414 *significand* **is** *digit-string* . [*digit-string*]
21 **or** . *digit-string*

22 R415 *exponent-letter* **is** E
23 **or** D

- 1 R416 *exponent* is *signed-digit-string*
- 2 C412 (R413) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.
- 3 C413 (R413) The value of *kind-param* shall specify an approximation method that exists on the
4 processor.
- 5 A real literal constant without a kind type parameter is a default real constant if it is without an
6 exponent part or has exponent letter E, and is a double precision real constant if it has exponent letter
7 D. A real literal constant written with a kind type parameter is a real constant with the specified kind
8 type parameter.
- 9 The exponent represents the power of ten scaling to be applied to the significand or digit string. The
10 meaning of these constants is as in decimal scientific notation.
- 11 The significand may be written with more digits than a processor will use to approximate the value of
12 the constant.

NOTE 4.10

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 4.4.4 Complex type

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

17 Each approximation method used to represent data entities of type real shall be available for both the
18 real and imaginary parts of a data entity of type complex. A **kind** type parameter may be specified for
19 a complex entity and selects for both parts the real approximation method characterized by this kind
20 type parameter value; this kind type parameter is of type default integer. The kind type parameter of
21 an approximation method is returned by the intrinsic inquiry function KIND (13.7.96).

22 The type specifier for the complex type uses the keyword COMPLEX. There is no keyword for double
23 precision complex. If the type keyword COMPLEX is specified and the kind type parameter is not
24 specified, the default kind value is the same as that for default real, the type of both parts is default
25 real, and the type specified is **default complex**.

- 26 R417 *complex-literal-constant* is (*real-part* , *imag-part*)
- 27 R418 *real-part* is *signed-int-literal-constant*
28 or *signed-real-literal-constant*
29 or *named-constant*
- 30 R419 *imag-part* is *signed-int-literal-constant*

```

1      or signed-real-literal-constant
2      or named-constant

```

3 C414 (R417) Each named constant in a complex literal constant shall be of type integer or real.

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

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

NOTE 4.11

Examples of complex literal constants are:

```
(1.0, -1.0)
(3, 3.1E6)
(4.0_4, 3.6E7_8)
( 0., PI)
```

where PI is a previously declared named real constant.

14 4.4.5 Character type

15 4.4.5.1 Character sets

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

20 The processor shall provide one or more **representation methods** that define sets of values for data
21 of type character. Each such method is characterized by a value for a type parameter called the **kind**
22 type parameter; this kind type parameter is of type default integer. The kind type parameter of a rep-
23 resentation method is returned by the intrinsic inquiry function KIND (13.7.96). The intrinsic function
24 SELECTED_CHAR_KIND (13.7.152) returns a kind value based on the name of a character type. Any
25 character of a particular representation method representable in the processor may occur in a character
26 string of that representation method.

27 The character set defined by ISO/IEC 646:1991 (International Reference Version) is referred to as the
28 **ASCII character set** and its corresponding representation method is the **ASCII character type**.
29 The character set defined by ISO/IEC 10646-1:2000 UCS-4 is referred to as the **ISO 10646 character**
30 **set** and its corresponding representation method is the **ISO 10646 character type**.

31 4.4.5.2 Character type specifier

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

33 If the kind type parameter is not specified, the default kind value is KIND ('A') and the type specified
34 is **default character**.

1 The default character kind shall support a character set that includes the Fortran character set. By
 2 supplying nondefault character kinds, the processor may support additional character sets. The charac-
 3 ters available in nondefault character kinds are not specified by this part of ISO/IEC 1539, except that
 4 one character in each nondefault character set shall be designated as a blank character to be used as a
 5 padding character.

6 R420 *char-selector* is *length-selector*
 7 or (LEN = *type-param-value* , ■
 8 ■ KIND = *scalar-int-initialization-expr*)
 9 or (*type-param-value* , ■
 10 ■ [KIND =] *scalar-int-initialization-expr*)
 11 or (KIND = *scalar-int-initialization-expr* ■
 12 ■ [, LEN = *type-param-value*])

13 R421 *length-selector* is ([LEN =] *type-param-value*)
 14 or * *char-length* [,]

15 R422 *char-length* is (*type-param-value*)
 16 or *int-literal-constant*

17 C415 (R420) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a rep-
 18 resentation method that exists on the processor.

19 C416 (R422) The *int-literal-constant* shall not include a *kind-param*.

20 C417 (R422) A *type-param-value* in a *char-length* shall be a colon, asterisk, or *specification-expr*.

21 C418 (R420 R421 R422) A *type-param-value* of * shall be used only

- 22 (1) to declare a dummy argument,
- 23 (2) to declare a named constant,
- 24 (3) in the *type-spec* of an ALLOCATE statement wherein each *allocate-object* is a dummy
- 25 argument of type CHARACTER with an assumed character length,
- 26 (4) in the *type-spec* or *derived-type-spec* of a type guard statement (8.1.9), or
- 27 (5) in an external function, to declare the character length parameter of the function result.

28 C419 A function name shall not be declared with an asterisk *type-param-value* unless it is of type CHAR-
 29 ACTER and is the name of the result of an external function or the name of a dummy function.

30 C420 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, elemental, recursive,
 31 or pure.

32 C421 (R421) The optional comma in a *length-selector* is permitted only in a *declaration-type-spec* in a *type-declaration-*
 33 *stmt*.

34 C422 (R421) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the
 35 *type-declaration-stmt*.

36 C423 (R420) The length specified for a character statement function or for a statement function dummy argument of
 37 type character shall be an initialization expression.

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

1 If the character length parameter value evaluates to a negative value, the length of character entities
2 declared is zero. A character length parameter value of : indicates a deferred type parameter (4.2). A
3 *char-length* type parameter value of * has the following meanings.

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

15 4.4.5.3 Character literal constant

16 A **character literal constant** is written as a sequence of characters, delimited by either apostrophes
17 or quotation marks.

18	R423	<i>char-literal-constant</i>	is	[<i>kind-param</i> _]	,	[<i>rep-char</i>]	...	,
19			or	[<i>kind-param</i> _]	"	[<i>rep-char</i>]	...	"

20 C424 (R423) The value of *kind-param* shall specify a representation method that exists on the pro-
21 cessor.

22 The optional kind type parameter preceding the leading delimiter specifies the kind type parameter of
23 the character constant; if it is not present, the constant is of type default character.

24 For the type character with kind *kind-param*, if present, and for type default character otherwise, a
25 **representable character**, *rep-char*, is defined as follows.

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

NOTE 4.12

FORTRAN 77 allowed any character to occur in a character context. This part of ISO/IEC 1539 allows a source program to contain characters of more than one kind. Some processors may identify characters of nondefault kinds by control characters (called “escape” or “shift” characters). It is difficult, if not impossible, to process, edit, and print files where some occurrences of control characters have their intended meaning and some occurrences might not. Almost all control characters have uses or effects that effectively preclude their use in character contexts and this is why free source form allows only graphic characters as representable characters. Nevertheless, for compatibility with FORTRAN 77, control characters remain permitted in principle in fixed source form.

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

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

35 A zero-length character literal constant is represented by two consecutive apostrophes (without inter-

- 1 vening blanks) or two consecutive quotation marks (without intervening blanks) outside of a character
 2 context.
- 3 The intrinsic operation **concatenation** (//) is defined between two data entities of type character
 4 (7.1.5.3) with the same kind type parameter.

NOTE 4.13

Examples of character literal constants are:

```
"DON'T"
'DON'T'
```

both of which have the value DON'T and

```
''
```

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

NOTE 4.14

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

```
NIHONGO_彼女なしでは何もできない。
```

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

5 4.4.5.4 Collating sequence

- 6 The processor defines a collating sequence for the character set of each kind of character. A **collating**
 7 **sequence** is a one-to-one mapping of the characters into the nonnegative integers such that each charac-
 8 ter corresponds to a different nonnegative integer. The intrinsic functions CHAR (13.7.30) and ICHAR
 9 (13.7.84) provide conversions between the characters and the integers according to this mapping.

NOTE 4.15

For example:

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

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

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

1 and lower-case letters.

2 The collating sequence for the ASCII character type is as defined by ISO/IEC 646:1991 (International
3 Reference Version); this collating sequence is called the **ASCII collating sequence** in this part of
4 ISO/IEC 1539. The collating sequence for the ISO 10646 character type is as defined by ISO/IEC
5 10646-1:2000.

NOTE 4.16

The intrinsic functions ACHAR (13.7.2) and IACHAR (13.7.77) provide conversion between characters and corresponding integer values according to the ASCII collating sequence.

6 The intrinsic functions LGT, LGE, LLE, and LLT (13.7.101-13.7.104) provide comparisons between
7 strings based on the ASCII collating sequence. International portability is guaranteed if the set of
8 characters used is limited to the letters, digits, underscore, and special characters.

9 4.4.6 Logical type

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

11 The processor shall provide one or more **representation methods** for data of type logical. Each such
12 method is characterized by a value for a type parameter called the **kind** type parameter; this kind type
13 parameter is of type default integer. The kind type parameter of a representation method is returned
14 by the intrinsic inquiry function KIND (13.7.96).

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

16 If the kind type parameter is not specified, the default kind value is KIND (.FALSE.) and the type
17 specified is **default logical**.

18 R424 *logical-literal-constant* **is** .TRUE. [*- kind-param*]
19 **or** .FALSE. [*- kind-param*]

20 C425 (R424) The value of *kind-param* shall specify a representation method that exists on the pro-
21 cessor.

22 The optional kind type parameter specifies the kind type parameter of the logical constant; if it is not
23 present, the constant is of type default logical.

24 The intrinsic operations defined for data entities of logical type are negation (.NOT.), conjunction
25 (.AND.), inclusive disjunction (.OR.), logical equivalence (.EQV.), and logical nonequivalence (.NEQV.,
26 .XOR.) as described in 7.1.5.4. There is also a set of intrinsically defined relational operators that
27 compare the values of data entities of other types and yield a value of type default logical. These
28 operations are described in 7.1.5.6.

29 4.4.7 Bits type

30 The **bits type** has a set of values composed of ordered sequences of bits. The number of bits in
31 the sequence is specified by the **kind** type parameter, which shall be greater than or equal to zero.
32 The processor shall provide **representation methods** with kind type parameter values equal to every
33 nonnegative integer less than or equal to a processor-determined limit. This limit shall be at least as
34 large as the storage size, expressed in bits, of every supported kind of type integer, real, complex, and
35 logical. Additional representation methods may be provided.

36 The type specifier for the bits type uses the keyword BITS.

37 If the kind type parameter is not specified for a bits variable, the default kind value is the size of a

1 numeric storage unit expressed in bits, and the type specified is **default bits**.

2 R425 *bits-literal-constant* **is** *binary-constant* [*_ kind-param*]
 3 **or** *octal-constant* [*_ kind-param*]
 4 **or** *hex-constant* [*_ kind-param*]

5 R426 *binary-constant* **is** B ' *digit* [*digit*] ... '
 6 **or** B " *digit* [*digit*] ... "

7 C426 (R426) *digit* shall have one of the values 0 or 1.

8 R427 *octal-constant* **is** O ' *digit* [*digit*] ... '
 9 **or** O " *digit* [*digit*] ... "

10 C427 (R427) *digit* shall have one of the values 0 through 7.

11 R428 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
 12 **or** Z " *hex-digit* [*hex-digit*] ... "

13 R429 *hex-digit* **is** *digit*
 14 **or** A
 15 **or** B
 16 **or** C
 17 **or** D
 18 **or** E
 19 **or** F

20 The *hex-digits* A through F represent the numbers ten through fifteen, respectively; they may be repre-
 21 sented by their lower-case equivalents.

22 If the optional kind type parameter is not specified for a bits literal constant, the kind value is assumed
 23 from the form of the constant. If the constant is a *binary-constant* the kind value is the number
 24 of *digit* characters. If the constant is an *octal-constant* the kind value is three times the number of
 25 *digit* characters. If the constant is a *hex-constant* the kind value is four times the number of *hex-digit*
 26 characters.

NOTE 4.17

Even if a bits value is too large to fit into a single statement as a literal constant, it can be constructed by concatenation of bits named constants.

27 Each digit of an octal constant represents three bits, and each hex digit of a hex constant represents
 28 four bits, according to their numerical representations as binary integers, with leading zero bits where
 29 needed.

30 If a *kind-param* is specified for a bits literal constant and has a value greater than the number of bits
 31 specified by its digits, the constant is padded on the left (13.3) with enough zero bits to create a constant
 32 of kind *kind-param*. If the *kind-param* specified has a value smaller the number of bits specified by its
 33 digits, only the rightmost *kind-param* bits are used to determine the value of the constant and the
 34 remaining bits shall be zero.

NOTE 4.18

Though the processor is required to provide bit kinds only up to four times the size of a numeric storage unit, or up to the maximum intrinsic type size (whichever is larger), it is expected that the actual size limit will be much larger, based on system capacity constraints. Use of BITS objects with KIND values equal to small integer multiples of NUMERIC_STORAGE_SIZE should result in more efficient execution.

4.5 Derived types

4.5.1 Derived type concepts

Additional types may be derived from the intrinsic types and other derived types. A type definition is required to define the name of the type and the names and attributes of its components and type-bound procedures.

A derived type may be parameterized by multiple type parameters, each of which is defined to be either a kind or length type parameter and may have a default value.

The **ultimate components** of an object of derived type are the components that are of intrinsic type or have the POINTER or ALLOCATABLE attribute, plus the ultimate components of the components of the object that are of derived type and have neither the ALLOCATABLE nor POINTER attribute.

NOTE 4.19

The ultimate components of objects of the derived type `kids` defined below are `name`, `age`, and `other_kids`.

```
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
```

The **direct components** of an object of derived type are the components of that object, plus the direct components of the components of the object that are of derived type and have neither the ALLOCATABLE nor POINTER attribute.

By default, no storage sequence is implied by the order of the component definitions. However, a storage order is implied for a sequence type (4.5.2.3). If the derived type has the BIND attribute, the storage sequence is that required by the companion processor (2.5.10, 15.3.4).

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

4.5.2 Derived-type definition

4.5.2.1 Syntax

R430	<i>derived-type-def</i>	is	<i>derived-type-stmt</i> [<i>type-param-def-stmt</i>] ... [<i>private-or-sequence</i>] ... [<i>component-part</i>] [<i>type-bound-procedure-part</i>] <i>end-type-stmt</i>
R431	<i>derived-type-stmt</i>	is	TYPE [[, <i>type-attr-spec-list</i>] ::] <i>type-name</i> ■ ■ [(<i>type-param-name-list</i>)]
R432	<i>type-attr-spec</i>	is	ABSTRACT

		or <i>access-spec</i>
		or BIND (C)
		or EXTENDS (<i>parent-type-name</i>)
C428	(R431)	A derived type <i>type-name</i> shall not be DOUBLEPRECISION or the same as the name of any intrinsic type defined in this part of ISO/IEC 1539.
C429	(R431)	The same <i>type-attr-spec</i> shall not appear more than once in a given <i>derived-type-stmt</i> .
C430	(R432)	A <i>parent-type-name</i> shall be the name of a previously defined extensible type (4.5.7).
C431	(R430)	If the type definition contains or inherits (4.5.7.2) a deferred binding (4.5.5), ABSTRACT shall appear.
C432	(R430)	If ABSTRACT appears, the type shall be extensible.
C433	(R430)	If EXTENDS appears, SEQUENCE shall not appear.
C434	(R430)	If EXTENDS appears and the type being defined has a co-array ultimate component, its parent type shall have a co-array ultimate component.
R433	<i>private-or-sequence</i>	is <i>private-components-stmt</i> or <i>sequence-stmt</i>
C435	(R430)	The same <i>private-or-sequence</i> shall not appear more than once in a given <i>derived-type-def</i> .
R434	<i>end-type-stmt</i>	is END TYPE [<i>type-name</i>]
C436	(R434)	If END TYPE is followed by a <i>type-name</i> , the <i>type-name</i> shall be the same as that in the corresponding <i>derived-type-stmt</i> .
Derived types with the BIND attribute are subject to additional constraints as specified in 15.3.4.		

NOTE 4.20

An example of a derived-type definition is:

```

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

```

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

TYPE (PERSON) :: CHAIRMAN

4.5.2.2 Accessibility

Types that are defined in a module or accessible in that module by use association have either the PUBLIC or PRIVATE attribute. Types for which an *access-spec* is not explicitly specified in that module have the default accessibility attribute for that module. The default accessibility attribute for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.4.1). Only types that have the PUBLIC attribute in that module are available to be accessed from that module by use association.

The accessibility of a type does not affect, and is not affected by, the accessibility of its components and bindings.

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

NOTE 4.21

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.

3 4.5.2.3 Sequence type

4 R435 *sequence-stmt* is SEQUENCE

5 C437 (R439) If SEQUENCE appears, each data component shall be declared to be of an intrinsic type
 6 or of a sequence derived type.

7 C438 (R430) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.

8 If the **SEQUENCE statement** appears, the type is a **sequence type**. The order of the component
 9 definitions in a sequence type specifies a storage sequence for objects of that type. The type is a
 10 **numeric sequence type** if there are no type parameters, no pointer or allocatable components, and
 11 each component is of type default integer, default real, double precision real, default complex, default
 12 logical, default bits, or a numeric sequence type. The type is a **character sequence type** if there
 13 are no type parameters, no pointer or allocatable components, and each component is of type default
 14 character or a character sequence type.

NOTE 4.22

An example of a numeric sequence type is:

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

NOTE 4.23

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

NOTE 4.23 (cont.)

as best suited to the particular architecture.

1 **4.5.2.4 Determination of derived types**

- 2 Derived-type definitions with the same type name may appear in different scoping units, in which case
 3 they may be independent and describe different derived types or they may describe the same type.
- 4 Two data entities have the same type if they are declared with reference to the same derived-type
 5 definition. The definition may be accessed from a module or from a host scoping unit. Data entities in
 6 different scoping units also have the same type if they are declared with reference to different derived-type
 7 definitions that specify the same type name, all have the SEQUENCE property or all have the BIND
 8 attribute, have no components with PRIVATE accessibility, and have type parameters and components
 9 that agree in order, name, and attributes. Otherwise, they are of different derived types. A data entity
 10 declared using a type with the SEQUENCE property or with the BIND attribute is not of the same type
 11 as an entity of a type declared to be PRIVATE or that has any components that are PRIVATE.

NOTE 4.24

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

```

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

```

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

NOTE 4.25

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

```

NOTE 4.25 (cont.)

```

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 property, 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 4.26

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

1 4.5.3 Derived-type parameters

2 4.5.3.1 Type parameter definition statement

3 R436 *type-param-def-stmt* **is** INTEGER [*kind-selector*] , *type-param-attr-spec* :: ■
4 ■ *type-param-decl-list*

5 R437 *type-param-decl* **is** *type-param-name* [= *scalar-int-initialization-expr*]

6 C439 (R436) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the
7 *type-param-names* in the *derived-type-stmt* of that *derived-type-def*.

8 C440 (R436) Each *type-param-name* in the *derived-type-stmt* in a *derived-type-def* shall appear as a
9 *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.

10 R438 *type-param-attr-spec* **is** KIND
11 **or** LEN

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

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

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

17 If a *type-param-decl* has a *scalar-int-initialization-expr*, the type parameter has a default value which
18 is specified by the expression. If necessary, the value is converted according to the rules of intrinsic
19 assignment (7.2.1.3) to a value of the same kind as the type parameter.

20 A type parameter may be used as a primary in a specification expression (7.1.11) in the *derived-type-*
21 *def*. A kind type parameter may also be used as a primary in an initialization expression (7.1.12) in the
22 *derived-type-def*.

NOTE 4.27

The following example uses derived-type parameters.

```

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

```

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

```

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

```

1 4.5.3.2 Type parameter order

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

4 The type parameter order of a nonextended type is the order of the type parameter list in the derived-
5 type definition. The type parameter order of an extended type (4.5.7 consists of the type parameter
6 order of its parent type followed by any additional type parameters in the order of the type parameter
7 list in the derived-type definition.

NOTE 4.28

Given

```

TYPE :: t1(k1,k2)
  INTEGER,KIND :: k1,k2
  REAL(k1) a(k2)
END TYPE
TYPE,EXTENDS(t1) :: t2(k3)
  INTEGER,KIND :: k3
  LOGICAL(k3) flag
END TYPE

```

the type parameter order for type T1 is K1 then K2, and the type parameter order for type T2 is K1 then K2 then K3.

8 4.5.4 Components

9 4.5.4.1 Component definition statement

10	R439	<i>component-part</i>	is	[<i>component-def-stmt</i>] ...
11	R440	<i>component-def-stmt</i>	is	<i>data-component-def-stmt</i>
12			or	<i>proc-component-def-stmt</i>
13	R441	<i>data-component-def-stmt</i>	is	<i>declaration-type-spec</i> [[, <i>component-attr-spec-list</i>] ::] ■
14				■ <i>component-decl-list</i>

- 1 R442 *component-attr-spec* is *access-spec*
 2 or ALLOCATABLE
 3 or DIMENSION (*component-array-spec*)
 4 or DIMENSION [(*deferred-shape-spec-list*)] ■
 5 ■ *lbracket co-array-spec rbracket*
 6 or CONTIGUOUS
 7 or POINTER
- 8 R443 *component-decl* is *component-name* [(*component-array-spec*)] ■
 9 ■ [*lbracket co-array-spec rbracket*] ■
 10 ■ [* *char-length*] [*component-initialization*]
- 11 R444 *component-array-spec* is *explicit-shape-spec-list*
 12 or *deferred-shape-spec-list*
 13
- 14 C441 (R441) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.
- 15 C442 (R441) If neither the POINTER nor ALLOCATABLE attribute is specified, the *declaration-type-spec*
 16 in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived
 17 type.
- 18 C443 (R441) If the POINTER or ALLOCATABLE attribute is specified, the *declaration-type-spec* in
 19 the *component-def-stmt* shall be CLASS(*) or shall specify an intrinsic type or any accessible
 20 derived type including the type being defined.
- 21 C444 (R441) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec*
 22 shall be a *deferred-shape-spec-list*.
- 23 C445 (R441) If a *co-array-spec* appears, it shall be a *deferred-co-shape-spec-list* and the component
 24 shall have the ALLOCATABLE attribute.
- 25 C446 (R441) If a *co-array-spec* appears, the component shall not be of type IMAGE_TEAM (13.8.2.7),
 26 C_PTR, or C_FUNPTR (15.3.3).
- 27 C447 A data component whose type has a co-array ultimate component shall be a nonpointer nonal-
 28 locatable scalar and shall not be a co-array.
- 29 C448 (R441) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each
 30 *component-array-spec* shall be an *explicit-shape-spec-list*.
- 31 C449 (R444) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a
 32 specification expression that does not contain references to specification functions or any object
 33 designators other than named constants or subobjects thereof.

J3 internal note

Unresolved Technical Issue 094

This constraint is inconsistent (as is the one 3 below): it allows

INTEGER c1(DIGITS(yvariable))

INTEGER c2(len_type_param)

but not

INTEGER c3(DIGITS(yvariable)+len_type_param)

It's not quite trivial to word this correctly...

- 1 C450 (R441) A component shall not have both the ALLOCATABLE and the POINTER attribute.
- 2 C451 (R441) If the CONTIGUOUS attribute is specified, the component shall be an array with the
3 POINTER attribute.
- 4 C452 (R443) The * *char-length* option is permitted only if the component is of type character.
- 5 C453 (R440) Each *type-param-value* within a *component-def-stmt* shall either be a colon, be an ini-
6 tialization expression, or be a specification expression that contains neither references to speci-
7 fication functions nor any object designators other than named constants or subobjects thereof.

NOTE 4.29

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

- 8 R445 *proc-component-def-stmt* is PROCEDURE ([*proc-interface*]) , ■
9 ■ *proc-component-attr-spec-list* :: *proc-decl-list*

NOTE 4.30

See 12.4.3.6 for definitions of *proc-interface* and *proc-decl*.

- 10 R446 *proc-component-attr-spec* is POINTER
11 or PASS [(*arg-name*)]
12 or NOPASS
13 or *access-spec*
- 14 C454 (R445) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-*
15 *component-def-stmt*.
- 16 C455 (R445) POINTER shall appear in each *proc-component-attr-spec-list*.
- 17 C456 (R445) If the procedure pointer component has an implicit interface or has no arguments,
18 NOPASS shall be specified.
- 19 C457 (R445) If PASS (*arg-name*) appears, the interface shall have a dummy argument named *arg-*
20 *name*.
- 21 C458 (R445) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.

4.5.4.2 Array components

23 A data component is an array if its *component-decl* contains a *component-array-spec* or its *data-compo-*
24 *nent-def-stmt* contains the DIMENSION clause with a *component-array-spec*. If the *component-decl*
25 contains a *component-array-spec*, it specifies the array rank, and if the array is explicit shape (5.3.7.2),
26 the array bounds; otherwise, the *component-array-spec* in the DIMENSION clause specifies the array
27 rank, and if the array is explicit shape, the array bounds.

28 A data component is a co-array if its *component-decl* contains a *co-array-spec* or its *data-component-def-*
29 *stmt* contains a DIMENSION clause with a *co-array-spec*. If the *component-decl* contains a *co-array-spec*
30 it specifies the co-rank; otherwise, the *co-array-spec* in the DIMENSION clause specifies the co-rank.

NOTE 4.31

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

NOTE 4.31 (cont.)

```

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

```

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

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

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

NOTE 4.32

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

```

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

```

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

NOTE 4.33

Default initialization of an explicit-shape array component may be specified by an initialization expression consisting of an array constructor (4.7), or of a single scalar that becomes the value of each array element.

1 4.5.4.3 Pointer components

- 2 A component is a pointer (2.4.7) if its *component-attr-spec-list* contains the `POINTER` attribute. A
 3 pointer component may be a data pointer or a procedure pointer.

NOTE 4.34

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

```

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

```

Any object of type `REFERENCE` will have the four nonpointer components `VOLUME`, `YEAR`, `PAGE`, and `TITLE`, the procedure pointer `PRINT`, which has an explicit interface the same as `printer_interface`, plus a pointer to an array of characters holding `SYNOPSIS`. The size of this

NOTE 4.34 (cont.)

target array will be determined by the length of the abstract. The space for the target may be allocated (6.3.1) or the pointer component may be associated with a target by a pointer assignment statement (7.2.2).

1 **4.5.4.4 The passed-object dummy argument**

2 A **passed-object dummy argument** is a distinguished dummy argument of a procedure pointer
 3 component or type-bound procedure. It affects procedure overriding (4.5.7.3) and argument association
 4 (12.5.2.2).

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

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

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

11 C459 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data
 12 object with the same declared type as the type being defined; all of its length type parameters
 13 shall be assumed; it shall be polymorphic (4.3.1.3) if and only if the type being defined is
 14 extensible (4.5.7). It shall not have the VALUE attribute.

NOTE 4.35

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.

15 **4.5.4.5 Default initialization for components**

16 Default initialization provides a means of automatically initializing pointer components to be disassoci-
 17 ated or associated with specific targets, and nonpointer nonallocatable components to have a particular
 18 value. Allocatable components are always initialized to not allocated.

19 A pointer variable or component is **data-pointer-initialization compatible** with a target if the pointer
 20 is type compatible with the target, they have the same rank, and the values of corresponding nondeferred
 21 type parameters are specified by initialization expressions that have the same value.

22 R447 *component-initialization* **is** = *initialization-expr*
 23 **or** => *null-init*
 24 **or** => *initial-data-target*

25 R448 *initial-data-target* **is** *designator*

26 C460 (R441) If *component-initialization* appears, a double-colon separator shall appear before the
 27 *component-decl-list*.

28 C461 (R441) If *component-initialization* appears, every type parameter and array bound of the com-
 29 ponent shall be a colon or initialization expression.

30 C462 (R441) If => appears in *component-initialization*, POINTER shall appear in the *component-*
 31 *attr-spec-list*. If = appears in *component-initialization*, neither POINTER nor ALLOCATABLE
 32 shall appear in the *component-attr-spec-list*.

- 1 C463 (R447) If *initial-data-target* appears, *component-name* shall be data-pointer-initialization com-
 2 patible with it.
- 3 C464 (R448) The *designator* shall designate a variable that is an initialization target. Every subscript,
 4 section subscript, substring starting point, and substring ending point in *designator* shall be an
 5 initialization expression.
- 6 If *null-init* appears for a pointer component, that component in any object of the type has an initial
 7 association status of disassociated (16.5.2.2) or becomes disassociated as specified in 16.5.2.2.2.
- 8 A variable is an **initialization target** if it has the TARGET attribute, either has the SAVE attribute
 9 or is declared in the main program, and does not have the ALLOCATABLE attribute.
- 10 If *initial-data-target* appears for a data pointer component, that component in any object of the type is
 11 initially associated with the target or becomes associated with the target as specified in 16.5.2.2.1.
- 12 If *initial-proc-target* (12.4.3.6) appears in *proc-decl* for a procedure pointer component, that component
 13 in any object of the type is initially associated with the target or becomes associated with the target as
 14 specified in 16.5.2.2.1.
- 15 If *initialization-expr* appears for a nonpointer component, that component in any object of the type
 16 is initially defined (16.6.3) or becomes defined as specified in 16.6.5 with the value determined from
 17 *initialization-expr*. If necessary, the value is converted according to the rules of intrinsic assignment
 18 (7.2.1.3) to a value that agrees in type, type parameters, and shape with the component. If the compo-
 19 nent is of a type for which default initialization is specified for a component, the default initialization
 20 specified by *initialization-expr* overrides the default initialization specified for that component. When
 21 one initialization **overrides** another it is as if only the overriding initialization were specified (see Note
 22 4.37). Explicit initialization in a type declaration statement (5.2) overrides default initialization (see
 23 Note 4.36). Unlike explicit initialization, default initialization does not imply that the object has the
 24 SAVE attribute.
- 25 A subcomponent (6.1.2) is **default-initialized** if the type of the object of which it is a component
 26 specifies default initialization for that component, and the subcomponent is not a subobject of an object
 27 that is default-initialized or explicitly initialized.

NOTE 4.36

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 = 1994      ! Partial default initialization
END TYPE DATE

```

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

```

TYPE (DATE), PARAMETER :: TODAY = DATE (21, "Feb.", 1995)

```

NOTE 4.37

The default initial value of a component of derived type may be overridden by default initialization specified in the definition of the type. Continuing the example of Note 4.36:

```

TYPE SINGLE_SCORE

```

NOTE 4.37 (cont.)

```

TYPE (DATE) :: PLAY_DAY = TODAY
INTEGER SCORE
TYPE (SINGLE_SCORE), POINTER :: NEXT => NULL ( )
END TYPE SINGLE_SCORE
TYPE (SINGLE_SCORE) SETUP

```

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

NOTE 4.38

Arrays of structures may be declared with elements that are partially or totally initialized by default. Continuing the example of Note 4.37 :

```

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 ("I. Manage",1,5,NULL ( ))

```

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

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

```

ALLOCATE (ORGANIZER % HISTORY)    ! A partially initialized object of type
                                  ! SINGLE_SCORE is created.

```

NOTE 4.39

A pointer component of a derived type may have as its target an object of that derived type. The type definition may 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 may be used to construct linked lists of objects of type NODE. See C.1.4 for an example. Linked lists can also be constructed using allocatable components.

NOTE 4.40

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

```

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

```

NOTE 4.40 (cont.)

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

1 **4.5.4.6 Component order**

2 **Component order** is an ordering of the nonparent components of a derived type; it is used for intrinsic
3 formatted input/output and structure constructors (where component keywords are not used). Parent
4 components are excluded from the component order of an extended type (4.5.7).

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

NOTE 4.41

Given the same type definitions as in Note 4.28, 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.

9 **4.5.4.7 Component accessibility**

10 R449 *private-components-stmt* is PRIVATE

11 C465 (R449) A *private-components-stmt* is permitted only if the type definition is within the speci-
12 fication part of a module.

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

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

NOTE 4.42

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

NOTE 4.43

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

NOTE 4.44

An example of a type with private components is:

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

NOTE 4.44 (cont.)

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

NOTE 4.45

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

```

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

TYPE (MIXED) :: M

```

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

4.5.5 Type-bound procedures

R450 *type-bound-procedure-part* is *contains-stmt*

[*binding-private-stmt*]

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

R451 *binding-private-stmt* is PRIVATE

C466 (R450) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.

R452 *type-bound-proc-binding* is *type-bound-procedure-stmt*

or *type-bound-generic-stmt*

or *final-procedure-stmt*

R453 *type-bound-procedure-stmt* is PROCEDURE [[, *binding-attr-list*] ::] ■

■ *binding-name* [=> *procedure-name*]

or PROCEDURE (*interface-name*) ■

■ , *binding-attr-list* :: *binding-name*

C467 (R453) If => *procedure-name* appears, the double-colon separator shall appear.

C468 (R453) The *procedure-name* shall be the name of an accessible module procedure or an external procedure that has an explicit interface.

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

R454 *type-bound-generic-stmt* is GENERIC ■

■ [, *access-spec*] :: *generic-spec* => *binding-name-list*

C469 (R454) 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.

- 1 C470 (R454) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the
2 type.
- 3 C471 (R454) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object
4 dummy argument (4.5.4.4).
- 5 C472 (R454) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall
6 be as specified in 12.4.3.4.2.
- 7 C473 (R454) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified
8 in 12.4.3.4.3.
- 9 C474 (R454) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in
10 9.5.4.7. The type of the **dtv** argument shall be *type-name*.
- 11 R455 *binding-attr* **is** PASS [(*arg-name*)]
12 **or** NOPASS
13 **or** NON_OVERRIDABLE
14 **or** DEFERRED
15 **or** *access-spec*
- 16 C475 (R455) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.
- 17 C476 (R453) If the interface of the binding has no dummy argument of the type being defined,
18 NOPASS shall appear.
- 19 C477 (R453) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument
20 named *arg-name*.
- 21 C478 (R455) PASS and NOPASS shall not both appear in the same *binding-attr-list*.
- 22 C479 (R455) NON_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-*
23 *attr-list*.
- 24 C480 (R455) DEFERRED shall appear if and only if *interface-name* appears.
- 25 C481 (R453) An overriding binding (4.5.7.3) shall have the DEFERRED attribute only if the binding
26 it overrides is deferred.
- 27 C482 (R453) A binding shall not override an inherited binding (4.5.7.2) that has the NON_OVER-
28 RIDABLE attribute.
- 29 A **type-bound procedure statement** declares a specific **type-bound procedure**. A specific type-
30 bound procedure may have a passed-object dummy argument (4.5.4.4). A binding that specifies the
31 DEFERRED attribute is a **deferred binding**. A deferred binding shall appear only in the definition
32 of an abstract type.
- 33 A **GENERIC statement** declares a type-bound generic interface for its specific type-bound procedures.
- 34 A **binding** of a type is a specific type-bound procedure, a generic type-bound interface, or a final
35 subroutine. These are referred to as specific bindings, generic bindings, and final bindings respectively.
- 36 A type-bound procedure may be identified by a **binding name** in the scope of the type definition.
37 This name is the *binding-name* for a specific binding, and the *generic-name* for a generic binding whose
38 *generic-spec* is *generic-name*. A final binding, or a generic binding whose *generic-spec* is not *generic-*
39 *name*, has no binding name.
- 40 The interface of a specific binding is that of the procedure specified by *procedure-name* or the interface

- 1 specified by *interface-name*.

NOTE 4.46

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
```

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

NOTE 4.47

Unlike the situation with generic procedure names, a generic type-bound procedure name is not permitted to be the same as a specific type-bound procedure name in the same type (16.3).

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

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

NOTE 4.48

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

10 4.5.6 Final subroutines

11 4.5.6.1 Declaration

12 R456 *final-procedure-stmt* **is** FINAL [::] *final-subroutine-name-list*

13 C483 (R456) A *final-subroutine-name* shall be the name of a module procedure with exactly one
 14 dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocat-
 15 able, nonpolymorphic variable of the derived type being defined. All length type parameters of
 16 the dummy argument shall be assumed. The dummy argument shall not have INTENT(OUT).

17 C484 (R456) A *final-subroutine-name* shall not be one previously specified as a final subroutine for
 18 that type.

1 C485 (R456) A final subroutine shall not have a dummy argument with the same kind type parameters
2 and rank as the dummy argument of another final subroutine of that type.

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

5 A derived type is **finalizable** if it has any final subroutines or if it has any nonpointer, nonallocatable
6 component whose type is finalizable. A nonpointer data entity is finalizable if its type is finalizable.

NOTE 4.49

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

NOTE 4.50

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.

7 4.5.6.2 The finalization process

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

- 10 (1) If the dynamic type of the entity has a final subroutine whose dummy argument has the
11 same kind type parameters and rank as the entity being finalized, it is called with the entity
12 as an actual argument. Otherwise, if there is an elemental final subroutine whose dummy
13 argument has the same kind type parameters as the entity being finalized, it is called with
14 the entity as an actual argument. Otherwise, no subroutine is called at this point.
- 15 (2) All finalizable components that appear in the type definition are finalized in a processor-
16 dependent order. If the entity being finalized is an array, each finalizable component of each
17 element of that entity is finalized separately.
- 18 (3) If the entity is of extended type and the parent type is finalizable, the parent component is
19 finalized.

20 If several entities are to be finalized as a consequence of an event specified in 4.5.6.3, the order in which
21 they are finalized is processor-dependent. A final subroutine shall not reference or define an object that
22 has already been finalized.

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

24 4.5.6.3 When finalization occurs

25 When a pointer is deallocated its target is finalized. When an allocatable entity is deallocated, it is
26 finalized.

27 A nonpointer, nonallocatable object that is not a dummy argument or function result is finalized im-
28 mediately before it would become undefined due to execution of a RETURN or END statement (16.6.6,
29 item (3)).

30 A nonpointer nonallocatable local variable of a BLOCK construct is finalized immediately before it
31 would become undefined due to termination of the BLOCK construct (16.6.6, item (20)).

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

- 1 If an executable construct references a structure constructor or array constructor, the entity created by
- 2 the constructor is finalized after execution of the innermost executable construct containing the reference.
- 3 If a specification expression in a scoping unit references a function, the result is finalized before execution
- 4 of the executable constructs in the scoping unit.
- 5 If a specification expression in a scoping unit references a structure constructor or array constructor, the
- 6 entity created by the constructor is finalized before execution of the executable constructs in the scoping
- 7 unit.
- 8 When a procedure is invoked, a nonpointer, nonallocatable object that is an actual argument associated
- 9 with an INTENT(OUT) dummy argument is finalized.
- 10 When an intrinsic assignment statement is executed, the variable is finalized after evaluation of *expr*
- 11 and before the definition of the variable.

NOTE 4.51

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

- 12 If an object is allocated via pointer allocation and later becomes unreachable due to all pointers to that
- 13 object having their pointer association status changed, it is processor dependent whether it is finalized.
- 14 If it is finalized, it is processor dependent as to when the final subroutines are called.

15 4.5.6.4 Entities that are not finalized

- 16 If image execution is terminated, either by an error (e.g. an allocation failure) or by execution of a STOP
- 17 or END PROGRAM statement, entities existing immediately prior to termination are not finalized.

NOTE 4.52

A nonpointer, nonallocatable object that has the SAVE attribute or that is declared in the main program, a module, or a submodule is never finalized as a direct consequence of the execution of a RETURN or END statement.

A variable in a module or submodule is not finalized if it retains its definition status and value, even when there is no active procedure referencing the module or submodule.

18 4.5.7 Type extension**19 4.5.7.1 Concepts**

- 20 A nonsequence derived type that does not have the BIND attribute is an **extensible type**.
- 21 An extensible type that does not have the EXTENDS attribute is a **base type**. A type that has the
- 22 EXTENDS attribute is an **extended type**. The **parent type** of an extended type is the type named
- 23 in the EXTENDS attribute specification.

NOTE 4.53

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

- 24 A base type is an **extension type** of itself only. An extended type is an extension of itself and of all
- 25 types for which its parent type is an extension.
- 26 An **abstract type** is a type that has the ABSTRACT attribute.

NOTE 4.54

A deferred binding (4.5.5) defers the implementation of a type-bound procedure to extensions of the type; it may appear only in an abstract type. The dynamic type of an object cannot be abstract; therefore, a deferred binding cannot be invoked. An extension of an abstract type need not be abstract if it has no deferred bindings. 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.1.3.

1 4.5.7.2 Inheritance

2 An extended type includes all of the type parameters, all of the components, and the nonoverridden
 3 (4.5.7.3) nonfinal procedure bindings of its parent type. These are **inherited** by the extended type from
 4 the parent type. They retain all of the attributes that they had in the parent type. Additional type
 5 parameters, components, and procedure bindings may be declared in the derived-type definition of the
 6 extended type.

NOTE 4.55

Inaccessible components and bindings of the parent type are also inherited, but they remain inaccessible in the extended type. Inaccessible entities occur if the type being extended is accessed via use association and has a private entity.

NOTE 4.56

A base 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.

7 An extended type has a scalar, nonpointer, nonallocatable, **parent component** with the type and
 8 type parameters of the parent type. The name of this component is the parent type name. It has the
 9 accessibility of the parent type. Components of the parent component are **inheritance associated**
 10 (16.5.4) with the corresponding components inherited from the parent type. An **ancestor component**
 11 of a type is the parent component of the type or an ancestor component of the parent component.

NOTE 4.57

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

NOTE 4.58

Examples:

```

TYPE POINT                                ! A base type
    REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT      ! An extension of TYPE(POINT)
    ! Components X and Y, and component name POINT, inherited from parent
    INTEGER :: COLOR

```

NOTE 4.58 (cont.)

END TYPE COLOR_POINT

1 4.5.7.3 Type-bound procedure overriding

2 If a nongeneric binding specified in a type definition has the same binding name as a binding from the
3 parent type then the binding specified in the type definition **overrides** the one from the parent type.

4 The overriding binding and the overridden binding shall satisfy the following conditions.

- 5 (1) Either both shall have a passed-object dummy argument or neither shall.
- 6 (2) If the overridden binding is pure then the overriding binding shall also be pure.
- 7 (3) Either both shall be elemental or neither shall.
- 8 (4) They shall have the same number of dummy arguments.
- 9 (5) Passed-object dummy arguments, if any, shall correspond by name and position.
- 10 (6) Dummy arguments that correspond by position shall have the same names and characteris-
11 tics, except for the type of the passed-object dummy arguments.
- 12 (7) Either both shall be subroutines or both shall be functions having the same result charac-
13 teristics (12.3.3).
- 14 (8) If the overridden binding is PUBLIC then the overriding binding shall not be PRIVATE.

NOTE 4.59

The following is an example of procedure overriding, expanding on the example in Note 4.46.

```
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
```

15 If a generic binding specified in a type definition has the same *generic-spec* as an inherited binding, it
16 extends the generic interface and shall satisfy the requirements specified in 12.4.3.4.5.

17 A binding of a type and a binding of an extension of that type correspond if the latter binding is the
18 same binding as the former, overrides a corresponding binding, or is an inherited corresponding binding.

19 4.5.8 Derived-type values

1 The **component value** of

- 2 (1) a pointer component is its pointer association,
- 3 (2) an allocatable component is its allocation status and, if it is allocated, its dynamic type and
- 4 type parameters, bounds and value, and
- 5 (3) a nonpointer nonallocatable component is its value.

6 The set of values of a particular derived type consists of all possible sequences of the component values
7 of its components.

8 **4.5.9 Derived-type specifier**

9 A derived-type specifier is used in several contexts to specify a particular derived type and type param-
10 eters.

11 R457 *derived-type-spec* **is** *type-name* [(*type-param-spec-list*)]

12 R458 *type-param-spec* **is** [*keyword* =] *type-param-value*

13 C486 (R457) *type-name* shall be the name of an accessible derived type.

14 C487 (R457) *type-param-spec-list* shall appear only if the type is parameterized.

15 C488 (R457) There shall be at most one *type-param-spec* corresponding to each parameter of the type.
16 If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding
17 to that type parameter.

18 C489 (R458) The *keyword*= may be omitted from a *type-param-spec* only if the *keyword*= has been
19 omitted from each preceding *type-param-spec* in the *type-param-spec-list*.

20 C490 (R458) Each *keyword* shall be the name of a parameter of the type.

21 C491 (R458) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the decla-
22 ration of a dummy argument or associate name or in the allocation of a dummy argument.

23 Type parameter values that do not have type parameter keywords specified correspond to type param-
24 eters in type parameter order (4.5.3.2). If a type parameter keyword is present, the value is assigned to
25 the type parameter named by the keyword. If necessary, the value is converted according to the rules of
26 intrinsic assignment (7.2.1.3) to a value of the same kind as the type parameter.

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

28 **4.5.10 Construction of derived-type values**

29 A derived-type definition implicitly defines a corresponding **structure constructor** that allows con-
30 struction of values of that derived type. The type and type parameters of a constructed value are
31 specified by a derived type specifier.

32 R459 *structure-constructor* **is** *derived-type-spec* ([*component-spec-list*])

33 R460 *component-spec* **is** [*keyword* =] *component-data-source*

34 R461 *component-data-source* **is** *expr*
35 **or** *data-target*
36 **or** *proc-target*

37 C492 (R459) The *derived-type-spec* shall not specify an abstract type (4.5.7).

- 1 C493 (R459) At most one *component-spec* shall be provided for a component.
- 2 C494 (R459) If a *component-spec* is provided for an ancestor component, a *component-spec* shall not
3 be provided for any component that is inheritance associated with a subcomponent of that
4 ancestor component.
- 5 C495 (R459) A *component-spec* shall be provided for a nonallocatable component unless it has default
6 initialization or is inheritance associated with a subcomponent of another component for which
7 a *component-spec* is provided.
- 8 C496 (R460) The *keyword=* may be omitted from a *component-spec* only if the *keyword=* has been
9 omitted from each preceding *component-spec* in the constructor.
- 10 C497 (R460) Each *keyword* shall be the name of a component of the type.
- 11 C498 (R459) The type name and all components of the type for which a *component-spec* appears shall
12 be accessible in the scoping unit containing the structure constructor.
- 13 C499 (R459) If *derived-type-spec* is a type name that is the same as a generic name, the *component-*
14 *spec-list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a
15 generic reference to that name (12.5.5.2).
- 16 C4100 (R461) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall
17 correspond to a procedure pointer component.
- 18 C4101 (R461) A *data-target* shall have the same rank as its corresponding component.

NOTE 4.60

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*.

- 19 In the absence of a component keyword, each *component-data-source* is assigned to the corresponding
20 component in component order (4.5.4.6). If a component keyword appears, the *expr* is assigned to the
21 component named by the keyword. For a nonpointer component, the declared type and type parameters
22 of the component and *expr* shall conform in the same way as for a *variable* and *expr* in an intrinsic
23 assignment statement (7.2.1.2), as specified in Table 7.12. If necessary, each value of intrinsic type is
24 converted according to the rules of intrinsic assignment (7.2.1.3) to a value that agrees in type and type
25 parameters with the corresponding component of the derived type. For a nonpointer nonallocatable
26 component, the shape of the expression shall conform with the shape of the component.
- 27 If a component with default initialization has no corresponding *component-data-source*, then the default
28 initialization is applied to that component. If an allocatable component has no corresponding *component-*
29 *data-source*, then that component has an allocation status of unallocated.

NOTE 4.61

Because no parent components appear in the defined component ordering, a value for a parent component can be specified only with a component keyword. Examples of equivalent values using types defined in Note 4.58:

```
! 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)
```


NOTE 4.61 (cont.)

COLOR_POINT(1, 2, 3)	! has private components ! All components of TYPE(point) ! need to be accessible.

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

NOTE 4.62

This example illustrates a derived-type constant expression using a derived type defined in Note 4.20:

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

This could also be written as

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

NOTE 4.63

An example constructor using the derived type GENERAL_POINT defined in Note 4.27 is

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

- 2 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 (7.2.2). If the component data source is
 4 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 4.64

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

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

and BIBLIO were declared using the derived-type definition REFERENCE in Note 4.34

```
TYPE (REFERENCE) :: BIBLIO
```

the statement

```
BIBLIO = REFERENCE (1, 1987, 1, "This is the title of the referenced &  

&paper", TEXT)
```

is valid and associates the pointer component SYNOPSIS of the object BIBLIO with the target object TEXT.

- 6 If a component of a derived type is allocatable, the corresponding constructor expression shall either be
 7 a reference to the intrinsic function NULL with no arguments, an allocatable entity of the same rank,
 8 or shall evaluate to an entity of the same rank. If the expression is a reference to the intrinsic function
 9 NULL, the corresponding component of the constructor has a status of unallocated. If the expression
 10 is an allocatable entity, the corresponding component of the constructor has the same allocation status
 11 as that allocatable entity and, if it is allocated, the same dynamic type, bounds, and value; if a length
 12 parameter of the component is deferred, its value is the same as the corresponding parameter of the

1 expression. Otherwise the corresponding component of the constructor has an allocation status of
2 allocated and has the same bounds and value as the expression.

NOTE 4.65

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

3 4.5.11 Derived-type operations and assignment

4 Intrinsic assignment of derived-type entities is described in 7.2.1. This part of ISO/IEC 1539 does
5 not specify any intrinsic operations on derived-type entities. Any operation on derived-type entities
6 or defined assignment (7.2.1.4) for derived-type entities shall be defined explicitly by a function or a
7 subroutine, and a generic interface (4.5.2, 12.4.3.2).

8 4.6 Enumerations and enumerators

9 An enumeration is a set of enumerators. An enumerator is a named integer constant. An enumeration
10 definition specifies the enumeration and its set of enumerators of the corresponding integer kind.

```

11  R462  enum-def          is  enum-def-stmt
12                                     enumerator-def-stmt
13                                     [ enumerator-def-stmt ] ...
14                                     end-enum-stmt

```

```

15  R463  enum-def-stmt      is  ENUM, BIND(C)

```

16 R464 *enumerator-def-stmt* is ENUMERATOR [::] *enumerator-list*

17 R465 *enumerator* **is** *named-constant* [= *scalar-int-initialization-expr*]

```
18  R466  end-enum-stmt      is  END ENUM
```

19 C4102 (R464) If = appears in an *enumerator*, a double-colon separator shall appear before the *enu-*
20 *enumerator-list*.

21 For an enumeration, the kind is selected such that an integer type with that kind is interoperable (15.3.2)
22 with the corresponding C enumeration type. The corresponding C enumeration type is the type that
23 would be declared by a C enumeration specifier (6.7.2.2 of the C International Standard) that specified
24 C enumeration constants with the same values as those specified by the *enum-def*, in the same order as
25 specified by the *enum-def*.

26 The companion processor (2.5.10) shall be one that uses the same representation for the types declared
27 by all C enumeration specifiers that specify the same values in the same order.

NOTE 4.66

If a companion processor uses an unsigned type to represent a given enumeration type, the Fortran processor will use the signed integer type of the same width for the enumeration, even though some of the values of the enumerators cannot be represented in this signed integer type. The types of any such enumerators will be interoperable with the type declared in the C enumeration.

NOTE 4.67

The C International Standard guarantees the enumeration constants fit in a C int (6.7.2.2 of the C International Standard). 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

NOTE 4.67 (cont.)

integer type that is interoperable with the corresponding C enumerated type.

NOTE 4.68

The C International Standard specifies that two enumeration types are compatible only if they specify enumeration constants with the same names and same values in the same order. This part of ISO/IEC 1539 further requires that a C processor that is to be a companion processor of a Fortran processor use the same representation for two enumeration types if they both specify enumeration constants with the same values in the same order, even if the names are different.

1 An enumerator is treated as if it were explicitly declared with the `PARAMETER` attribute. The enu-
 2 merator is defined in accordance with the rules of intrinsic assignment (7.2) with the value determined
 3 as follows.

- 4 (1) If *scalar-int-initialization-expr* is specified, the value of the enumerator is the result of
 5 *scalar-int-initialization-expr*.
- 6 (2) If *scalar-int-initialization-expr* is not specified and the enumerator is the first enumerator
 7 in *enum-def*, the enumerator has the value 0.
- 8 (3) If *scalar-int-initialization-expr* is not specified and the enumerator is not the first enumer-
 9 ator in *enum-def*, its value is the result of adding 1 to the value of the enumerator that
 10 immediately precedes it in the *enum-def*.

NOTE 4.69

Example of an enumeration definition:

```
ENUM, BIND(C)
  ENUMERATOR :: RED = 4, BLUE = 9
  ENUMERATOR YELLOW
END ENUM
```

The kind type parameter for this enumeration is processor dependent, but the processor is required to select a kind sufficient to represent the values 4, 9, and 10, which are the values of its enumerators. The following declaration might be equivalent to the above enumeration definition.

```
INTEGER(SELECTED_INT_KIND(2)), PARAMETER :: RED = 4, BLUE = 9, YELLOW = 10
```

An entity of the same kind type parameter value can be declared using the intrinsic function `KIND` with one of the enumerators as its argument, for example

```
INTEGER(KIND(RED)) :: X
```

NOTE 4.70

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.

11 4.7 Construction of array values

12 An **array constructor** is defined as a sequence of scalar values and is interpreted as a rank-one array where the element values are those specified in the sequence.

- 1 R467 *array-constructor* **is** (*/ ac-spec /*)
2 **or** *lbracket ac-spec rbracket*
- 3 R468 *ac-spec* **is** *type-spec ::*
4 **or** [*type-spec ::*] *ac-value-list*
- 5 R469 *lbracket* **is** [
6 R470 *rbracket* **is**]
- 7 R471 *ac-value* **is** *expr*
8 **or** *ac-implied-do*
- 9 R472 *ac-implied-do* **is** (*ac-value-list* , *ac-implied-do-control*)
- 10 R473 *ac-implied-do-control* **is** *ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■
11 ■ [, *scalar-int-expr*]
- 12 R474 *ac-do-variable* **is** *do-variable*
- 13 C4103 (R468) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the
14 same type and kind type parameters.
- 15 C4104 (R468) If *type-spec* specifies an intrinsic type, each *ac-value* expression in the *array-constructor*
16 shall be of an intrinsic type that is in type conformance with a variable of type *type-spec* as
17 specified in Table 7.12.
- 18 C4105 (R468) If *type-spec* specifies a derived type, all *ac-value* expressions in the *array-constructor*
19 shall be of that derived type and shall have the same kind type parameter values as specified by
20 *type-spec*.
- 21 C4106 (R472) The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear
22 as the *ac-do-variable* of the containing *ac-implied-do*.
- 23 If *type-spec* is omitted, each *ac-value* expression in the array constructor shall have the same length type
24 parameters; in this case, the type and type parameters of the array constructor are those of the *ac-value*
25 expressions.
- 26 If *type-spec* appears, it specifies the type and type parameters of the array constructor. Each *ac-value*
27 expression in the *array-constructor* shall be compatible with intrinsic assignment to a variable of this
28 type and type parameters. Each value is converted to the type parameters of the *array-constructor* in
29 accordance with the rules of intrinsic assignment (7.2.1.3).
- 30 The character length of an *ac-value* in an *ac-implied-do* whose iteration count is zero shall not depend
31 on the value of the *ac-do-variable* and shall not depend on the value of an expression that is not an
32 initialization expression.
- 33 If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-*
34 *value* is an array expression, the values of the elements of the expression, in array element order (6.2.2.1),
35 specify the corresponding sequence of elements of the array constructor. If an *ac-value* is an *ac-implied-*
36 *do*, it is expanded to form a sequence of elements under the control of the *ac-do-variable*, as in the DO
37 construct (8.1.7.5).
- 38 For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct.
- 39 An empty sequence forms a zero-sized array.

NOTE 4.71

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.7.146). An example is:

```
X = (/ 3.2, 4.01, 6.5 /)
Y = RESHAPE (SOURCE = [ 2.0, [ 4.5, 4.5 ], X ], SHAPE = [ 3, 2 ])
```

This results in Y having the 3×2 array of values:

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

NOTE 4.72

Examples of array constructors containing an implied DO are:

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

and

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

NOTE 4.73

Using the type definition for PERSON in Note 4.20, an example of the construction of a derived-type array value is:

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

NOTE 4.74

Using the type definition for LINE in Note 4.31, an example of the construction of a derived-type scalar value with a rank-2 array component is:

```
LINE (RESHAPE ( [ 0.0, 0.0, 1.0, 2.0 ], [ 2, 2 ] ), 0.1, 1)
```

The RESHAPE intrinsic function is used to construct a value that represents a solid line from (0, 0) to (1, 2) of width 0.1 centimeters.

NOTE 4.75

Examples of zero-size array constructors are:

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

NOTE 4.76

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

NOTE 4.76 (cont.)

the constants with the same character length.

5 Attribute declarations and specifications

5.1 General

Every data object has a type and rank and may have type parameters and other properties that determine the uses of the object. Collectively, these properties are the **attributes** of the object. The type of a named data object is either specified explicitly in a type declaration statement or determined implicitly by the first letter of its name (5.5). All of its attributes may be specified in a type declaration statement or individually in separate specification statements.

A function has a type and rank and may have type parameters and other attributes that determine the uses of the function. The type, rank, and type parameters are the same as those of its result variable.

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

5.2 Type declaration statements

5.2.1 Syntax

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

The type declaration statement specifies the type of the entities in the entity declaration list. The type and type parameters are those specified by *declaration-type-spec*, except that the character length type parameter may be overridden for an entity by the appearance of * *char-length* in its *entity-decl*.

R502 *attr-spec* **is** *access-spec*
 or ALLOCATABLE
 or ASYNCHRONOUS
 or CONTIGUOUS
 or dimension-spec
 or EXTERNAL
 or INTENT (*intent-spec*)
 or INTRINSIC
 or *language-binding-spec*
 or OPTIONAL
 or PARAMETER
 or POINTER
 or PROTECTED
 or SAVE
 or TARGET
 or VALUE
 or VOLATILE

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

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

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

1 names.

2 The type declaration statement also specifies the attributes whose keywords appear in the *attr-spec*,
3 except that the DIMENSION attribute may be specified or overridden for an entity by the appearance
4 of *array-spec* in its *entity-decl*.

5 R503 *entity-decl* **is** *object-name* [(*array-spec*)] ■
6 ■ [*lbracket co-array-spec rbracket*] ■
7 ■ [* *char-length*] [*initialization*]
8 **or** *function-name* [* *char-length*]

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

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

11 C506 (R503) An *initialization* shall not appear if *object-name* is a dummy argument, a function result,
12 an object in a named common block unless the type declaration is in a block data program unit,
13 an object in blank common, an allocatable variable, an external function, an intrinsic function,
14 or an automatic object.

15 C507 (R503) An *initialization* shall appear if the entity is a named constant (5.3.12).

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

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

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

20 R505 *initialization* **is** = *initialization-expr*
21 **or** => *null-init*
22 **or** => *initial-data-target*

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

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

26 C511 (R503) If *initial-data-target* appears, *object-name* shall be data-pointer-initialization compatible
27 with it (4.5.4.5).

28 C512 (R506) The *function-reference* shall be a reference to the NULL intrinsic function with no
29 arguments.

30 A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.6. An
31 explicit type declaration statement is not required; however, it is permitted. Specifying a type for a
32 generic intrinsic function name in a type declaration statement is not sufficient, by itself, to remove the
33 generic properties from that function.

34 5.2.2 Automatic data objects

35 An **automatic data object** is a nondummy data object with a type parameter or array bound that
36 depends on the value of a *specification-expr* that is not an initialization expression.

37 C513 An automatic object shall not be a local variable of a main program, module, or submodule.

NOTE 5.1

An automatic object shall not have the SAVE attribute and shall not appear in a common block.

- 1 If a type parameter in a *declaration-type-spec* or in a *char-length* in an *entity-decl* is defined by an
 2 expression that is not an initialization expression, the type parameter value is established on entry to
 3 the procedure or BLOCK construct and is not affected by any redefinition or undefinition of the variables
 4 in the expression during execution of the procedure or BLOCK construct.

5.2.3 Initialization

- 6 The appearance of *initialization* in an *entity-decl* for an entity without the PARAMETER attribute
 7 specifies that the entity is a variable with **explicit initialization**. Explicit initialization alternatively
 8 may be specified in a DATA statement unless the variable is of a derived type for which default initial-
 9 ization is specified. If *initialization* is *=initialization-expr*, the variable is initially defined with the value
 10 specified by the *initialization-expr*; if necessary, the value is converted according to the rules of intrinsic
 11 assignment (7.2.1.3) to a value that agrees in type, type parameters, and shape with the variable. A
 12 variable, or part of a variable, shall not be explicitly initialized more than once in a program. If the
 13 variable is an array, it shall have its shape specified in either the type declaration statement or a previous
 14 attribute specification statement in the same scoping unit.
- 15 If *null-init* appears, the initial association status of the object is disassociated. If *initial-data-target*
 16 appears, the object is initially associated with the target.
- 17 Explicit initialization of a variable that is not in a common block implies the SAVE attribute, which
 18 may be confirmed by explicit specification.

5.2.4 Examples of type declaration statements**NOTE 5.2**

```

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.
REAL (KIND (0.0D0)) A
REAL (KIND = 2) B
COMPLEX (KIND = KIND (0.0D0)) :: C
CHARACTER (LEN = 10, KIND = 2) A
CHARACTER B, C *20
TYPE (PERSON) :: CHAIRMAN
TYPE(NODE), POINTER :: HEAD => NULL ( )
TYPE (humongous_matrix (k=8, d=1000)) :: mat

```

(The last line above uses a type definition from Note 4.27.)

5.3 Attributes**5.3.1 Constraints**

- 22 An attribute may be explicitly specified by an *attr-spec* in a type declaration statement or by an attribute
 specification statement (5.4). The following constraints apply to attributes.

1

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

3 C515 An *array-spec* for a function result that does not have the ALLOCATABLE or POINTER
4 attribute shall be an *explicit-shape-spec-list*.

5 C516 The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy
6 argument of a procedure that has a *proc-language-binding-spec*.

7 5.3.2 Accessibility attribute

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

```

9  R507    access-spec                is PUBLIC
10                                     or PRIVATE

```

11 C517 (R507) An *access-spec* shall appear only in the *specification-part* of a module.

12 Identifiers that are specified in a module or accessible in that module by use association have either the
13 **PUBLIC** or **PRIVATE** attribute. Identifiers for which an *access-spec* is not explicitly specified in
14 that module have the default accessibility attribute for that module. The default accessibility attribute
15 for a module is **PUBLIC** unless it has been changed by a **PRIVATE** statement (5.4.1). Only identifiers
16 that have the **PUBLIC** attribute in that module are available to be accessed from that module by use
17 association.

NOTE 5.3

In order for an identifier to be accessed by use association, it must have the PUBLIC attribute in the module from which it is accessed. It can nonetheless have the PRIVATE attribute in a module in which it is accessed by use association, and therefore not be available for use association from that module.

NOTE 5.4

An example of an accessibility specification is:

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

18 5.3.3 ALLOCATABLE attribute

19 An entity with the **ALLOCATABLE** attribute is a variable for which space is allocated by an AL-
20 LOCATE statement (6.3.1) or by an intrinsic assignment statement (7.2.1.3).

21 5.3.4 ASYNCHRONOUS attribute

22 An entity with the **ASYNCHRONOUS** attribute is a variable that may be subject to asynchronous
23 input/output.

24 The base object of a variable shall have the `ASYNCHRONOUS` attribute in a scoping unit if

- 25 (1) the variable appears in an executable statement or specification expression in that scoping
26 unit and
- 27 (2) any statement of the scoping unit is executed while the variable is a pending I/O storage
28 sequence affector (9.5.2.5).

29 Use of a variable in an asynchronous input/output statement can imply the ASYNCHRONOUS attribute;

- 1 see subclause (9.5.2.5).
- 2 An object may have the ASYNCHRONOUS attribute in a particular scoping unit without necessarily
- 3 having it in other scoping units (11.2.2, 16.5.1.4). If an object has the ASYNCHRONOUS attribute,
- 4 then all of its subobjects also have the ASYNCHRONOUS attribute.

NOTE 5.5

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.

The ASYNCHRONOUS attribute is similar to the VOLATILE attribute. It is intended to facilitate traditional code motion optimizations in the presence of asynchronous input/output.

5 5.3.5 BIND attribute for data entities

- 6 The BIND attribute for a variable or common block specifies that it is capable of interoperating with a
- 7 C variable that has external linkage (15.4).
- 8 R508 *language-binding-spec* is BIND (C [, NAME = *scalar-char-initialization-expr*])
- 9 C518 An entity with the BIND attribute shall be a common block, variable, or procedure.
- 10 C519 A variable with the BIND attribute shall be declared in the specification part of a module.
- 11 C520 A variable with the BIND attribute shall be interoperable (15.3).
- 12 C521 Each variable of a common block with the BIND attribute shall be interoperable.
- 13 C522 (R508) The *scalar-char-initialization-expr* shall be of default character kind. If the value of the
- 14 *scalar-char-initialization-expr* after discarding leading and trailing blanks has nonzero length,
- 15 it shall be valid as an identifier on the companion processor.

NOTE 5.6

The C International Standard provides a facility for creating C identifiers whose characters are not restricted to the C basic character set. Such a C identifier is referred to as a universal character name (6.4.3 of the C International Standard). The name of such a C identifier might include characters that are not part of the representation method used by the processor for type default character. If so, the C entity cannot be referenced from Fortran.

- 16 The BIND attribute for a variable or common block implies the SAVE attribute, which may be confirmed
- 17 by explicit specification.

18 5.3.6 CONTIGUOUS attribute

- 19 C523 An entity with the CONTIGUOUS attribute shall be an array pointer or an assumed-shape
- 20 array.
- 21 The **CONTIGUOUS attribute** specifies that an assumed-shape array can only be argument associated
- 22 with a contiguous effective argument, or that an array pointer can only be pointer associated with a
- 23 contiguous target.
- 24 An object is **contiguous** if it is
- 25 (1) an object with the CONTIGUOUS attribute,

- 1 (2) a nonpointer whole array that is not assumed-shape,
- 2 (3) an assumed-shape array that is argument associated with an array that is contiguous,
- 3 (4) an array allocated by an `ALLOCATE` statement,
- 4 (5) a pointer associated with a contiguous target, or
- 5 (6) a nonzero-sized array section (6.2.2) provided that
 - 6 (a) its base object is contiguous,
 - 7 (b) it does not have a vector subscript,
 - 8 (c) the elements of the section, in array element order, are a subset of the base object
 - 9 elements that are consecutive in array element order,
 - 10 (d) if the array is of type character and a *substring-range* appears, the *substring-range*
 - 11 specifies all of the characters of the *parent-string* (6.1.1),
 - 12 (e) only its final *part-ref* has nonzero rank, and
 - 13 (f) it is not the real or imaginary part (6.1.3) of an array of type complex.

14 An object is not contiguous if it is an array subobject, and

- 15 (1) the object has two or more elements,
- 16 (2) the elements of the object in array element order are not consecutive in the elements of the
- 17 base object,
- 18 (3) the object is not of type character with length zero, and
- 19 (4) the object is not of a derived type that has no ultimate components other than zero-sized
- 20 arrays and characters with length zero.

21 It is processor-dependent whether any other object is contiguous.

NOTE 5.7

If a derived type has only one component that is not zero-sized, it is processor-dependent whether a structure component of a contiguous array of that type is contiguous. That is, the derived type might contain padding on some processors.

NOTE 5.8

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
```

22 5.3.7 DIMENSION attribute

23 5.3.7.1 General

24 The **DIMENSION attribute** specifies that an entity is an array, a co-array, or both. If an *array-spec*

25 appears, it is an array. If a *co-array-spec* appears, it is a co-array.

26 For an array, its *array-spec* specifies its rank or rank and shape. For a co-array, its *co-array-spec* specifies

27 its co-rank or co-rank and co-bounds.

NOTE 5.9

Unless it is a dummy argument, a co-array has the same bounds and co-bounds on every image.

NOTE 5.9 (cont.)

See Note 12.32 for further discussion of the bounds and co-bounds of dummy co-arrays.

- 1 R509 *dimension-spec* **is** `DIMENSION ( array-spec )`
 2 **or** `DIMENSION [ ( array-spec ) ] lbracket co-array-spec rbracket`
- 3 C524 (R501) A co-array with the `ALLOCATABLE` attribute shall be specified with a *co-array-spec*
 4 that is a *deferred-co-shape-spec-list*.
- 5 C525 A co-array shall be a component or a variable that is not a function result.
- 6 C526 A co-array shall not be of type `IMAGE_TEAM` (13.8.2.7), `C_PTR`, or `C_FUNPTR` (15.3.3).
- 7 C527 An entity whose type has a co-array ultimate component shall be a nonpointer nonallocatable
 8 scalar, shall not be a co-array, and shall not be a function result.

NOTE 5.10

A co-array 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.

- 9 C528 The `SAVE` attribute shall be specified for a co-array unless it is a dummy argument, declared
 10 in the main program, or allocatable.

NOTE 5.11

This requirement for the `SAVE` attribute has the effect that automatic co-arrays are not permitted; for example, the co-array `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 co-arrays 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 co-arrays.

- 11 R510 *array-spec* **is** *explicit-shape-spec-list*
 12 **or** *assumed-shape-spec-list*
 13 **or** *deferred-shape-spec-list*
 14 **or** *assumed-size-spec*
 15 **or** *implied-shape-spec-list*
- 16 R511 *co-array-spec* **is** *deferred-co-shape-spec-list*
 17 **or** *explicit-co-shape-spec*
- 18 C529 The sum of the rank and co-rank of an entity shall not exceed fifteen.

NOTE 5.12

Examples of `DIMENSION` attribute specifications are:

```
SUBROUTINE EX (N, A, B)
```

NOTE 5.12 (cont.)

REAL, DIMENSION (N, 10) :: W	! Automatic explicit-shape array
REAL A (:), B (0:)	! Assumed-shape arrays
REAL, POINTER :: D (:, :)	! Array pointer
REAL, DIMENSION (:), POINTER :: P	! Array pointer
REAL, ALLOCATABLE, DIMENSION (:) :: E	! Allocatable array
REAL, PARAMETER :: V(0:*) = [0.1, 1.1]	! Implied-shape array

1 **5.3.7.2 Explicit-shape array**

2 An **explicit-shape array** is a named array that is declared with an *explicit-shape-spec-list*. This specifies
3 explicit values for the bounds in each dimension of the array.

4 R512 *explicit-shape-spec* **is** [*lower-bound* :] *upper-bound*

5 R513 *lower-bound* **is** *specification-expr*

6 R514 *upper-bound* **is** *specification-expr*

7 C530 (R512) An *explicit-shape-spec* whose bounds are not initialization expressions shall appear only
8 in a subprogram or interface body.

9 An explicit-shape array that is a local variable of a subprogram or BLOCK construct may have bounds
10 that are not initialization expressions. The bounds, and hence shape, are determined at entry to a
11 procedure defined by the subprogram, or on execution of the BLOCK statement, by evaluating the
12 bounds expressions. The bounds of such an array are unaffected by the redefinition or undefinition of
13 any variable during execution of the procedure or BLOCK construct.

14 The values of each *lower-bound* and *upper-bound* determine the bounds of the array along a particular
15 dimension and hence the extent of the array in that dimension. The value of a lower bound or an upper
16 bound may be positive, negative, or zero. The subscript range of the array in that dimension is the set
17 of integer values between and including the lower and upper bounds, provided the upper bound is not
18 less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the
19 extent in that dimension is zero, and the array is of zero size. If the *lower-bound* is omitted, the default
20 value is 1. The rank is equal to the number of *explicit-shape-specs*.

21 **5.3.7.3 Assumed-shape array**

22 An **assumed-shape array** is a nonallocatable nonpointer dummy argument array that takes its shape
23 from the associated actual argument array.

24 R515 *assumed-shape-spec* **is** [*lower-bound*] :

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

26 The extent of a dimension of an assumed-shape array dummy argument is the extent of the corresponding
27 dimension of the associated actual argument array. If the lower bound value is d and the extent of the
28 corresponding dimension of the associated actual argument array is s , then the value of the upper bound
29 is $s + d - 1$. If *lower-bound* appears it specifies the lower bound; otherwise the lower bound is 1.

30 **5.3.7.4 Deferred-shape array**

31 A **deferred-shape array** is an allocatable array or an array pointer.

32 An **allocatable array** is an array that has the ALLOCATABLE attribute and a specified rank, but its
bounds, and hence shape, are determined by allocation or argument association.

1

2 An **array pointer** is an array with the POINTER attribute and a specified rank. Its bounds, and hence
3 shape, are determined when it is associated with a target.

4 R516 *deferred-shape-spec* **is** :

5 C531 An array with the POINTER or ALLOCATABLE attribute shall have an *array-spec* that is a
6 *deferred-shape-spec-list*.

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

8 The size, bounds, and shape of an unallocated allocatable array or a disassociated array pointer are
9 undefined. No part of such an array shall be referenced or defined; however, the array may appear as an
10 argument to an intrinsic inquiry function as specified in 13.1.

11 The bounds of each dimension of an allocated allocatable array are those specified when the array
12 is allocated or, if it is a dummy argument, when it is argument associated with an allocated actual
13 argument.

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

- 15 (1) in an ALLOCATE statement (6.3.1) when the target is allocated,
- 16 (2) by pointer assignment (7.2.2), or
- 17 (3) if it is a dummy argument, by argument association with a nonpointer actual argument or
18 an associated pointer actual argument.

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

21 5.3.7.5 Assumed-size array

22 An **assumed-size array** is a dummy argument array whose size is assumed from that of an associated
23 actual argument. The rank and extents may differ for the actual and dummy arrays; only the size of the
24 actual array is assumed by the dummy array. An assumed-size array is declared with an *assumed-size-*
25 *spec*.

26 R517 *assumed-size-spec* **is** [*explicit-shape-spec* ,]... [*lower-bound* :] *

27 C532 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy
28 data argument.

29 C533 An assumed-size array with INTENT (OUT) shall not be polymorphic, of a finalizable type, of
30 a type with an ultimate allocatable component, or of a type for which default initialization is
31 specified.

32 The size of an assumed-size array is determined as follows.

- 33 (1) If the actual argument associated with the assumed-size dummy array is an array of any
34 type other than default character, the size is that of the actual array.
- 35 (2) If the actual argument associated with the assumed-size dummy array is an array element
36 of any type other than default character with a subscript order value of r (6.2.2.1) in an
37 array of size x , the size of the dummy array is $x - r + 1$.
- 38 (3) If the actual argument is a default character array, default character array element, or a
39 default character array element substring (6.1.1), and if it begins at character storage unit t
40 of an array with c character storage units, the size of the dummy array is MAX (INT $((c -$
41 $t + 1)/e)$, 0), where e is the length of an element in the dummy character array.

1 (4) If the actual argument is of type default character and is a scalar that is not an array element
 2 or array element substring designator, the size of the dummy array is MAX (INT (l/e), 0),
 3 where e is the length of an element in the dummy character array and l is the length of the
 4 actual argument.

5 The rank is equal to one plus the number of *explicit-shape-specs*.

6 An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last
 7 dimension and no shape. An assumed-size array name shall not be written as a whole array reference
 8 except as an actual argument in a procedure reference for which the shape is not required.

9 If a list of *explicit-shape-specs* appears, it specifies the bounds of the first rank–1 dimensions. If *lower-*
 10 *bound* appears it specifies the lower bound of the last dimension; otherwise that lower bound is 1. An
 11 assumed-size array may be subscripted or sectioned (6.2.2.2). The upper bound shall not be omitted
 12 from a subscript triplet in the last dimension.

13 If an assumed-size array has bounds that are not initialization expressions, the bounds are determined
 14 at entry to the procedure. The bounds of such an array are unaffected by the redefinition or undefinition
 15 of any variable during execution of the procedure.

16 **5.3.7.6 Implied-shape array**

17 An **implied-shape array** is a named constant that takes its shape from the *initialization-expr* in its
 18 declaration. An implied-shape array is declared with an *implied-shape-spec-list*.

19 R518 *implied-shape-spec* **is** [*lower-bound* :] *

20 C534 An implied-shape array shall be a named constant.

21 The rank of an implied-shape array is the number of *implied-shape-specs* in the *implied-shape-spec-list*.

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

25 **5.3.7.7 Allocatable co-array**

26 A co-array with the ALLOCATABLE attribute has a specified co-rank, but its co-bounds are determined
 27 by allocation or argument association.

28 R519 *deferred-co-shape-spec* **is** :

29 C535 A co-array with the ALLOCATABLE attribute shall have a *co-array-spec* that is a *deferred-co-*
 30 *shape-spec-list*.

31 The co-rank of an allocatable co-array is equal to the number of colons in its *deferred-co-shape-spec-list*.

32 The co-bounds of an unallocated allocatable co-array are undefined. No part of such a co-array shall be
 33 referenced or defined; however, the co-array may appear as an argument to an intrinsic inquiry function
 34 as specified in 13.1.

35 The co-bounds of an allocated allocatable co-array are those specified when the co-array is allocated.

36 The co-bounds of an allocatable co-array are unaffected by any subsequent redefinition or undefinition
 37 of the variables on which the bounds' expressions depend.

38 **5.3.7.8 Explicit-co-shape co-array**

1 An **explicit-co-shape co-array** is a named co-array that has its co-rank and co-bounds declared by
 2 an *explicit-co-shape-spec*.

3 R520 *explicit-co-shape-spec* is [[*lower-co-bound* :] *upper-co-bound*,]... ■
 4 ■ [*lower-co-bound* :] *

5 C536 A co-array that does not have the ALLOCATABLE attribute shall have a *co-array-spec* that is
 6 an *explicit-co-shape-spec*.

7 The co-rank is equal to one plus the number of *upper-co-bounds*.

8 R521 *lower-co-bound* is *specification-expr*

9 R522 *upper-co-bound* is *specification-expr*

10 C537 (R520) A *lower-co-bound* or *upper-co-bound* that is not an initialization expression shall appear
 11 only in a subprogram or interface body.

12 If an explicit-co-shape co-array has co-bounds that are not initialization expressions, the co-bounds are
 13 determined at entry to the procedure by evaluating the co-bounds expressions. The co-bounds of such
 14 a co-array are unaffected by the redefinition or undefinition of any variable during execution of the
 15 procedure.

16 The values of each *lower-co-bound* and *upper-co-bound* determine the co-bounds of the array along a
 17 particular co-dimension. The subscript range of the array in that co-dimension is the set of integer values
 18 between and including the lower and upper co-bounds. If the lower co-bound is omitted, the default
 19 value is 1. The upper co-bound shall not be less than the lower co-bound.

20 5.3.8 EXTERNAL attribute

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

23 C538 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

24 C539 In an external subprogram, the EXTERNAL attribute shall not be specified for a procedure
 25 defined by the subprogram.

26 If an external procedure or dummy procedure is used as an actual argument or is the target of a procedure
 27 pointer assignment, it shall be declared to have the EXTERNAL attribute.

28 A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

29 5.3.9 INTENT attribute

30 The **INTENT attribute** specifies the intended use of a dummy argument. An INTENT (IN) dummy
 31 argument is suitable for receiving data from the invoking scoping unit, an INTENT (OUT) dummy
 32 argument is suitable for returning data to the invoking scoping unit, and an INTENT (INOUT) dummy
 33 argument is suitable for use both to receive data from and to return data to the invoking scoping unit.

34 R523 *intent-spec* is IN
 35 or OUT
 36 or INOUT

37 C540 An entity with the INTENT attribute shall be a dummy data object or a dummy procedure
 38 pointer.

39 C541 (R523) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable

1 definition context (16.6.7).

2 C542 A pointer with the INTENT (IN) attribute shall not appear in a pointer association context
3 (16.6.8).

4 The INTENT (IN) attribute for a nonpointer dummy argument specifies that it shall neither be de-
5 fined nor become undefined during the execution of the procedure. The INTENT (IN) attribute for a
6 pointer dummy argument specifies that during the execution of the procedure its association shall not
7 be changed except that it may become undefined if the target is deallocated other than through the
8 pointer (16.5.2.2.3).

9 The INTENT (OUT) attribute for a nonpointer dummy argument specifies that the dummy argument
10 becomes undefined on invocation of the procedure, except for any subcomponents that are default-
11 initialized (4.5.4.5). Any actual argument that becomes associated with such a dummy argument shall
12 be definable. The INTENT (OUT) attribute for a pointer dummy argument specifies that on invocation
13 of the procedure the pointer association status of the dummy argument becomes undefined. Any actual
14 argument associated with such a pointer dummy shall be a pointer variable.

15 The INTENT (INOUT) attribute for a nonpointer dummy argument specifies that any actual argument
16 associated with the dummy argument shall be definable. The INTENT (INOUT) attribute for a pointer
17 dummy argument specifies that any actual argument associated with the dummy argument shall be a
18 pointer variable.

NOTE 5.13

The INTENT attribute for an allocatable dummy argument applies to both the allocation status and the definition status. An actual argument associated with an INTENT(OUT) allocatable dummy argument is deallocated on procedure invocation (6.3.3.1).

19 If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the
20 argument associated entity (12.5.2).

NOTE 5.14

An example of INTENT specification is:

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

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

NOTE 5.15

If a dummy argument is a derived-type object with a pointer component, then the pointer as a pointer is a subobject of the dummy argument, but the target of the pointer is not. Therefore, the restrictions on subobjects of the dummy object apply to the pointer in contexts where it is used as a pointer, but not in contexts where it is dereferenced to indicate its target. For example, if X is a dummy argument of derived type with an integer pointer component P, and X has INTENT(IN), then the statement

```
X%P => NEW_TARGET
```

is prohibited, but

NOTE 5.15 (cont.)

$X\%P = 0$

is allowed (provided that $X\%P$ is associated with a definable target).

Similarly, the **INTENT** restrictions on pointer dummy arguments apply only to the association of the dummy argument; they do not restrict the operations allowed on its target.

NOTE 5.16

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 should retain its current value rather than being redefined, **INTENT (INOUT)** should be used rather than **INTENT (OUT)**, 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 argument corresponding to a dummy argument without an **INTENT** attribute need be definable only if the dummy argument is actually redefined.

1 **5.3.10 INTRINSIC attribute**

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

4 If the specific name of an intrinsic procedure (13.6) is used as an actual argument, the name shall be
5 explicitly specified to have the **INTRINSIC** attribute. An intrinsic procedure whose specific name is
6 marked with a bullet (●) in 13.6 shall not be used as an actual argument.

7 C543 If the name of a generic intrinsic procedure is explicitly declared to have the **INTRINSIC** at-
8 tribute, and it is also the generic name of one or more generic interfaces (12.4.3.2) accessible in
9 the same scoping unit, the procedures in the interfaces and the specific intrinsic procedures shall
10 all be functions or all be subroutines, and the characteristics of the specific intrinsic procedures
11 and the procedures in the interfaces shall differ as specified in 12.4.3.4.5.

12 **5.3.11 OPTIONAL attribute**

13 The **OPTIONAL attribute** specifies that the dummy argument need not be associated with an actual
14 argument in a reference to the procedure (12.5.2.13).

15 C544 An entity with the **OPTIONAL** attribute shall be a dummy argument.

NOTE 5.17

The **PRESENT** intrinsic function (13.7.138) can be used to determine whether an optional dummy argument is associated with an actual argument.

1 **5.3.12 PARAMETER attribute**

2 The **PARAMETER attribute** specifies that an entity is a named constant. The entity has the value
3 specified by its *initialization-expr*, converted, if necessary, to the type, type parameters and shape of the
4 entity.

5 C545 An entity with the **PARAMETER** attribute shall not be a variable, a co-array, or a procedure.

6 A named constant shall not be referenced unless it has been defined previously in the same statement,
7 defined in a prior statement, or made accessible by use or host association.

NOTE 5.18

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 **5.3.13 POINTER attribute**

9 Entities with the **POINTER attribute** can be associated with different data objects or procedures
10 during execution of a program. A pointer is either a data pointer or a procedure pointer. Procedure
11 pointers are described in 12.4.3.6.

12 C546 An entity with the **POINTER** attribute shall not have the **ALLOCATABLE**, **INTRINSIC**, or
13 **TARGET** attribute, and shall not be a co-array.

14 C547 A procedure with the **POINTER** attribute shall have the **EXTERNAL** attribute.

15 A data pointer shall not be referenced unless it is pointer associated with a target object that is defined.

16 A data pointer shall not be defined unless it is pointer associated with a target object that is definable.

17 If a data pointer is associated, the values of its deferred type parameters are the same as the values of
18 the corresponding type parameters of its target.

19 A procedure pointer shall not be referenced unless it is pointer associated with a target procedure.

NOTE 5.19

Examples of **POINTER** attribute specifications are:

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

For a more elaborate example see C.2.1.

20 **5.3.14 PROTECTED attribute**

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

22 C548 The **PROTECTED** attribute shall be specified only in the specification part of a module.

- 1 C549 An entity with the PROTECTED attribute shall be a procedure pointer or variable.
- 2 C550 An entity with the PROTECTED attribute shall not be in a common block.
- 3 C551 A nonpointer object that has the PROTECTED attribute and is accessed by use association
4 shall not appear in a variable definition context (16.6.7) or as the *data-target* or *proc-target* in
5 a *pointer-assignment-stmt*.
- 6 C552 A pointer that has the PROTECTED attribute and is accessed by use association shall not
7 appear in a pointer association context (16.6.8).
- 8 Other than within the module in which an entity is given the PROTECTED attribute, or within any of
9 its descendants,
- 10 (1) if it is a nonpointer object, it is not definable, and
- 11 (2) if it is a pointer, its association status shall not be changed except that it may become
12 undefined if its target is deallocated other than through the pointer (16.5.2.2.3) or if its
13 target becomes undefined by execution of a RETURN or END statement.
- 14 If an object has the PROTECTED attribute, all of its subobjects have the PROTECTED attribute.

NOTE 5.20

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.

15 5.3.15 SAVE attribute

- 16 The SAVE attribute specifies that a local variable of a program unit or subprogram retains its association
17 status, allocation status, definition status, and value after execution of a RETURN or END statement
18 unless it is a pointer and its target becomes undefined (16.5.2.2.3(4)). If it is a local variable of a
19 subprogram it is shared by all instances (12.6.2.4) of the subprogram.
- 20 The SAVE attribute specifies that a local variable of a BLOCK construct retains its association status,
21 allocation status, definition status, and value after termination of the construct unless it is a pointer
22 and its target becomes undefined (16.5.2.2.3(5)). If the BLOCK construct is within a subprogram the
23 variable is shared by all instances (12.6.2.4) of the subprogram.
- 24 Giving a common block the SAVE attribute confers the attribute on all variables in the common block.
- 25 C553 An entity with the SAVE attribute shall be a common block, variable, or procedure pointer.
- 26 C554 The SAVE attribute shall not be specified for a dummy argument, a function result, an automatic
27 data object, or an object that is in a common block.

1 A **saved** entity is an entity that has the SAVE attribute. An **unsaved** entity is an entity that does not
 2 have the SAVE attribute.

3 The SAVE attribute has no effect on entities declared in a main program, module, or submodule. If
 4 a common block has the SAVE attribute in any other kind of scoping unit, it shall have the SAVE
 5 attribute in every scoping unit that is not a main program, module, or submodule.

6 **5.3.16 TARGET attribute**

7 The **TARGET attribute** specifies that a data object may have a pointer associated with it (7.2.2).
 8 An object without the TARGET attribute shall not have an accessible pointer associated with it.

9 C555 An entity with the TARGET attribute shall be a variable.

10 C556 An entity with the TARGET attribute shall not have the POINTER attribute.

NOTE 5.21

In addition to variables explicitly declared to have the TARGET attribute, the objects created by allocation of pointers (6.3.1.2) have the TARGET attribute.

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

NOTE 5.22

Examples of TARGET attribute specifications are:

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

For a more elaborate example see C.2.2.

NOTE 5.23

Every object designator that starts from a target object will have either the TARGET or POINTER attribute. If pointers are involved, the designator might not necessarily be a subobject of the original target object, but because pointers may point only to targets, there is no way to end up at a nonpointer that is not a target.

13 **5.3.17 VALUE attribute**

14 The **VALUE attribute** specifies a type of argument association (12.5.2.5) for a dummy argument.

15 C557 An entity with the VALUE attribute shall be a scalar dummy data object.

16 C558 An entity with the VALUE attribute shall not have the ALLOCATABLE, INTENT(INOUT),
 17 INTENT(OUT), POINTER, or VOLATILE attributes.

18 C559 If an entity has the VALUE attribute, any length type parameter value in its declaration shall
 19 be omitted or specified by an initialization expression.

20 **5.3.18 VOLATILE attribute**

21 The **VOLATILE attribute** specifies that an object may be referenced, defined, or become undefined,
 22 by means not specified by the program, or by another image without synchronization.

1 C560 An entity with the VOLATILE attribute shall be a variable that is not an INTENT(IN) dummy
2 argument.

3 An object may have the VOLATILE attribute in a particular scoping unit without necessarily having
4 it in other scoping units (11.2.2, 16.5.1.4). If an object has the VOLATILE attribute, then all of its
5 subobjects also have the VOLATILE attribute.

NOTE 5.24

The Fortran processor should use the most recent definition of a volatile object when a value is required. Likewise, it should make the most recent Fortran definition available. It is the programmer's responsibility to manage any interaction with non-Fortran processes.

6 A pointer with the VOLATILE attribute may additionally have its association status, dynamic type and
7 type parameters, and array bounds changed by means not specified by the program.

NOTE 5.25

If the target of a pointer is referenced, defined, or becomes undefined, by means not specified by the program, while the pointer is associated with the target, then the pointer shall have the VOLATILE attribute. Usually a pointer should have the VOLATILE attribute if its target has the VOLATILE attribute. Similarly, all members of an EQUIVALENCE group should have the VOLATILE attribute if one member has the VOLATILE attribute.

8 An allocatable object with the VOLATILE attribute may additionally have its allocation status, dynamic
9 type and type parameters, and array bounds changed by means not specified by the program.

5.4 Attribute specification statements

5.4.1 Accessibility statement

12 R524 *access-stmt* **is** *access-spec* [[::] *access-id-list*]

13 R525 *access-id* **is** *use-name*
14 **or** *generic-spec*

15 C561 (R524) An *access-stmt* shall appear only in the *specification-part* of a module. Only one ac-
16 cessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a
17 module.

18 C562 (R525) Each *use-name* shall be the name of a named variable, procedure, derived type, named
19 constant, namelist group, or macro.

20 An *access-stmt* with an *access-id-list* specifies the accessibility attribute (5.3.2), PUBLIC or PRIVATE,
21 of each *access-id* in the list. An *access-stmt* without an *access-id* list specifies the default accessibility
22 that applies to all potentially accessible identifiers in the *specification-part* of the module. The statement

23 PUBLIC

24 specifies a default of public accessibility. The statement

25 PRIVATE

26 specifies a default of private accessibility. If no such statement appears in a module, the default is public
27 accessibility.

NOTE 5.26

Examples of accessibility statements are:

```
MODULE EX
  PRIVATE
  PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)
```

1 5.4.2 ALLOCATABLE statement

2 R526 *allocatable-stmt* is ALLOCATABLE [::] *allocatable-decl-list*

3 R527 *allocatable-decl* is *object-name* [(*array-spec*)] ■
 4 ■ [*lbracket co-array-spec rbracket*]

5 The **ALLOCATABLE statement** specifies the ALLOCATABLE attribute (5.3.3) for a list of objects.

NOTE 5.27

An example of an ALLOCATABLE statement is:

```
REAL A, B (:), SCALAR
ALLOCATABLE :: A (:, :), B, SCALAR
```

6 5.4.3 ASYNCHRONOUS statement

7 R528 *asynchronous-stmt* is ASYNCHRONOUS [::] *object-name-list*

8 The **ASYNCHRONOUS statement** specifies the ASYNCHRONOUS attribute (5.3.4) for a list of
 9 objects.

10 5.4.4 BIND statement

11 R529 *bind-stmt* is *language-binding-spec* [::] *bind-entity-list*

12 R530 *bind-entity* is *entity-name*
 13 or / *common-block-name* /

14 C563 (R529) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of
 15 a single *bind-entity*.

16 The **BIND statement** specifies the BIND attribute (5.3.5) for a list of variables and common blocks.

17 5.4.5 CONTIGUOUS statement

18 R531 *contiguous-stmt* is CONTIGUOUS [::] *object-name-list*

19 The **CONTIGUOUS statement** specifies the CONTIGUOUS attribute (5.3.6) for a list of objects.

20 5.4.6 DATA statement

21 R532 *data-stmt* is DATA *data-stmt-set* [[,] *data-stmt-set*] ...

22 The **DATA statement** specifies explicit initialization (5.2.3).

23 A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If a
 24 nonpointer object has been specified with default initialization in a type definition, it shall not appear

1 in a *data-stmt-object-list*.

2 A variable that appears in a DATA statement and has not been typed previously may appear in a
3 subsequent type declaration only if that declaration confirms the implicit typing. An array name,
4 array section, or array element that appears in a DATA statement shall have had its array properties
5 established by a previous specification statement.

6 Except for variables in named common blocks, a named variable has the SAVE attribute if any part of
7 it is initialized in a DATA statement, and this may be confirmed by explicit specification.

8 R533 *data-stmt-set* **is** *data-stmt-object-list* / *data-stmt-value-list* /

9 R534 *data-stmt-object* **is** *variable*
10 **or** *data-implied-do*

11 R535 *data-implied-do* **is** (*data-i-do-object-list* , *data-i-do-variable* = ■
12 ■ *scalar-int-initialization-expr* , ■
13 ■ *scalar-int-initialization-expr* ■
14 ■ [, *scalar-int-initialization-expr*])

15 R536 *data-i-do-object* **is** *array-element*
16 **or** *scalar-structure-component*
17 **or** *data-implied-do*

18 R537 *data-i-do-variable* **is** *do-variable*

19 C564 A *data-stmt-object* or *data-i-do-object* shall not be a co-indexed variable.

20 C565 (R534) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring start-
21 ing point, and substring ending point shall be an initialization expression.

22 C566 (R534) A variable whose designator appears as a *data-stmt-object* or a *data-i-do-object* shall
23 not be a dummy argument, made accessible by use association or host association, in a named
24 common block unless the DATA statement is in a block data program unit, in a blank common
25 block, a function name, a function result name, an automatic object, or an allocatable variable.

26 C567 (R534) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object
27 designator in which a pointer appears other than as the entire rightmost *part-ref*.

28 C568 (R536) The *array-element* shall be a variable.

29 C569 (R536) The *scalar-structure-component* shall be a variable.

30 C570 (R536) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *sub-*
31 *script-list*.

32 C571 (R536) In an *array-element* or *scalar-structure-component* that is a *data-i-do-object*, any sub-
33 script shall be an initialization expression, and any primary within that subscript that is a
34 *data-i-do-variable* shall be a DO variable of this *data-implied-do* or of a containing *data-implied-*
35 *do*.

36 R538 *data-stmt-value* **is** [*data-stmt-repeat* *] *data-stmt-constant*

37 R539 *data-stmt-repeat* **is** *scalar-int-constant*
38 **or** *scalar-int-constant-subobject*

39 C572 (R539) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named con-
40 stant, it shall have been declared previously in the scoping unit or made accessible by use

1 association or host association.

2 R540 *data-stmt-constant* **is** *scalar-constant*
 3 **or** *scalar-constant-subobject*
 4 **or** *signed-int-literal-constant*
 5 **or** *signed-real-literal-constant*
 6 **or** *null-init*
 7 **or** *initial-data-target*
 8 **or** *structure-constructor*

9 C573 (R540) If a DATA statement constant value is a named constant or a structure constructor, the
 10 named constant or derived type shall have been declared previously in the scoping unit or made
 11 accessible by use or host association.

12 C574 (R540) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.

13 R541 *int-constant-subobject* **is** *constant-subobject*

14 C575 (R541) *int-constant-subobject* shall be of type integer.

15 R542 *constant-subobject* **is** *designator*

16 C576 (R542) *constant-subobject* shall be a subobject of a constant.

17 C577 (R542) Any subscript, substring starting point, or substring ending point shall be an initializa-
 18 tion expression.

19 The *data-stmt-object-list* is expanded to form a sequence of pointers and scalar variables, referred to
 20 as “sequence of variables” in subsequent text. A nonpointer array whose unqualified name appears
 21 as a *data-stmt-object* or *data-i-do-object* is equivalent to a complete sequence of its array elements in
 22 array element order (6.2.2.1). An array section is equivalent to the sequence of its array elements in
 23 array element order. A *data-implied-do* is expanded to form a sequence of array elements and structure
 24 components, under the control of the *data-i-do-variable*, as in the DO construct (8.1.7.5).

25 The *data-stmt-value-list* is expanded to form a sequence of *data-stmt-constants*. A *data-stmt-repeat*
 26 indicates the number of times the following *data-stmt-constant* is to be included in the sequence; omission
 27 of a *data-stmt-repeat* has the effect of a repeat factor of 1.

28 A zero-sized array or a *data-implied-do* with an iteration count of zero contributes no variables to the
 29 expanded sequence of variables, but a zero-length scalar character variable does contribute a variable
 30 to the expanded sequence. A *data-stmt-constant* with a repeat factor of zero contributes no *data-stmt-*
 31 *constants* to the expanded sequence of scalar *data-stmt-constants*.

32 The expanded sequences of variables and *data-stmt-constants* are in one-to-one correspondence. Each
 33 *data-stmt-constant* specifies the initial value, initial data target, or *null-init* for the corresponding vari-
 34 able. The lengths of the two expanded sequences shall be the same.

35 A *data-stmt-constant* shall be *null-init* or *initial-data-target* if and only if the corresponding *data-stmt-*
 36 *object* has the POINTER attribute. If *data-stmt-constant* is *null-init*, the initial association status of
 37 the corresponding data statement object is disassociated. If *data-stmt-constant* is *initial-data-target* the
 38 corresponding data statement object shall be data-pointer-initialization compatible with the initial data
 39 target; the data statement object is initially associated with the target.

40 A *data-stmt-constant* other than *null-init* or *initial-data-target* shall be compatible with its corresponding
 41 variable according to the rules of intrinsic assignment (7.2.1.2). The variable is initially defined with
 42 the value specified by the *data-stmt-constant*; if necessary, the value is converted according to the rules
 43 of intrinsic assignment (7.2.1.3) to a value that agrees in type, type parameters, and shape with the

1 variable.

NOTE 5.28

Examples of DATA statements are:

```
CHARACTER (LEN = 10)  NAME
INTEGER, DIMENSION (0:9) :: MILES
REAL, DIMENSION (100, 100) :: SKEW
TYPE (NODE), POINTER :: HEAD_OF_LIST
TYPE (PERSON) MYNAME, YOURNAME
DATA NAME / 'JOHN DOE' /, MILES / 10 * 0 /
DATA ((SKEW (K, J), J = 1, K), K = 1, 100) / 5050 * 0.0 /
DATA ((SKEW (K, J), J = K + 1, 100), K = 1, 99) / 4950 * 1.0 /
DATA HEAD_OF_LIST / NULL() /
DATA MYNAME / PERSON (21, 'JOHN SMITH') /
DATA YOURNAME % AGE, YOURNAME % NAME / 35, 'FRED BROWN' /
```

The character variable NAME is initialized with the value JOHN DOE with padding on the right because the length of the constant is less than the length of the variable. All ten elements of the integer array MILES are initialized to zero. The two-dimensional array SKEW is initialized so that the lower triangle of SKEW is zero and the strict upper triangle is one. The structures MYNAME and YOURNAME are declared using the derived type PERSON from Note 4.20. The pointer HEAD_OF_LIST is declared using the derived type NODE from Note 4.39; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

2 5.4.7 DIMENSION statement

3 R543 *dimension-stmt* **is** DIMENSION [::] *dimension-decl-list*
 4 R544 *dimension-decl* **is** *array-name* (*array-spec*)
 5 **or** *co-name* [(*array-spec*)] *lbracket co-array-spec rbracket*

6 The **DIMENSION** statement specifies the DIMENSION attribute (5.3.7) for a list of objects.

NOTE 5.29

An example of a DIMENSION statement is:

```
DIMENSION A (10), B (10, 70), C (:)
```

7 5.4.8 INTENT statement

8 R545 *intent-stmt* **is** INTENT (*intent-spec*) [::] *dummy-arg-name-list*

9 The **INTENT** statement specifies the INTENT attribute (5.3.9) for the dummy arguments in the list.

NOTE 5.30

An example of an INTENT statement is:

```
SUBROUTINE EX (A, B)
  INTENT (INOUT) :: A, B
```

10 5.4.9 OPTIONAL statement

1 R546 *optional-stmt* **is** OPTIONAL [::] *dummy-arg-name-list*

2 The **OPTIONAL statement** specifies the OPTIONAL attribute (5.3.11) for the dummy arguments
3 in the list.

NOTE 5.31

An example of an OPTIONAL statement is:

```
SUBROUTINE EX (A, B)
  OPTIONAL :: B
```

4 **5.4.10 PARAMETER statement**

5 The **PARAMETER statement** specifies the PARAMETER attribute (5.3.12) and the values for the
6 named constants in the list.

7 R547 *parameter-stmt* **is** PARAMETER (*named-constant-def-list*)

8 R548 *named-constant-def* **is** *named-constant* = *initialization-expr*

9 If a named constant is defined by a PARAMETER statement, it shall not be subsequently declared to
10 have a type or type parameter value that differs from the type and type parameters it would have if
11 declared implicitly (5.5). A named array constant defined by a PARAMETER statement shall have its
12 shape specified in a prior specification statement.

13 The value of each named constant is that specified by the corresponding initialization expression; if
14 necessary, the value is converted according to the rules of intrinsic assignment (7.2.1.3) to a value that
15 agrees in type, type parameters, and shape with the named constant.

NOTE 5.32

An example of a PARAMETER statement is:

```
PARAMETER (MODULUS = MOD (28, 3), NUMBER_OF_SENATORS = 100)
```

16 **5.4.11 POINTER statement**

17 R549 *pointer-stmt* **is** POINTER [::] *pointer-decl-list*

18 R550 *pointer-decl* **is** *object-name* [(*deferred-shape-spec-list*)]

19 **or** *proc-entity-name*

20 The **POINTER statement** specifies the POINTER attribute (5.3.13) for a list of entities.

NOTE 5.33

An example of a POINTER statement is:

```
TYPE (NODE) :: CURRENT
POINTER :: CURRENT, A (:, :)
```

21 **5.4.12 PROTECTED statement**

22 R551 *protected-stmt* **is** PROTECTED [::] *entity-name-list*

23 The **PROTECTED statement** specifies the PROTECTED attribute (5.3.14) for a list of entities.

1 5.4.13 SAVE statement

2 R552 *save-stmt* is SAVE [[::] *saved-entity-list*]

3 R553 *saved-entity* is *object-name*
 4 or *proc-pointer-name*
 5 or / *common-block-name* /

6 R554 *proc-pointer-name* is *name*

7 C578 (R552) If a SAVE statement with an omitted saved entity list appears in a scoping unit, no
 8 other appearance of the SAVE *attr-spec* or SAVE statement is permitted in that scoping unit.

9 A **SAVE statement** with a saved entity list specifies the SAVE attribute (5.3.15) for a list of entities.
 10 A SAVE statement without a saved entity list is treated as though it contained the names of all allowed
 11 items in the same scoping unit.

NOTE 5.34

An example of a SAVE statement is:

SAVE A, B, C, / BLOCKA /, D

12 5.4.14 TARGET statement

13 R555 *target-stmt* is TARGET [[::] *target-decl-list*

14 R556 *target-decl* is *object-name* [(*array-spec*)] ■

15 ■ [*lbracket co-array-spec rbracket*]

16 The **TARGET statement** specifies the TARGET attribute (5.3.16) for a list of objects.

NOTE 5.35

An example of a TARGET statement is:

TARGET :: A (1000, 1000), B

17 5.4.15 VALUE statement

18 R557 *value-stmt* is VALUE [[::] *dummy-arg-name-list*

19 The **VALUE statement** specifies the VALUE attribute (5.3.17) for a list of dummy arguments.

20 5.4.16 VOLATILE statement

21 R558 *volatile-stmt* is VOLATILE [[::] *object-name-list*

22 The **VOLATILE statement** specifies the VOLATILE attribute (5.3.18) for a list of objects.

23 5.5 IMPLICIT statement

24 In a scoping unit, an **IMPLICIT statement** specifies a type, and possibly type parameters, for all
 25 implicitly typed data entities whose names begin with one of the letters specified in the statement.
 26 Alternatively, it may indicate that no implicit typing rules are to apply in a particular scoping unit.

1 R559 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*
 2 **or** IMPLICIT NONE

3 R560 *implicit-spec* **is** *declaration-type-spec* (*letter-spec-list*)

4 R561 *letter-spec* **is** *letter* [– *letter*]

5 C579 (R559) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER
 6 statements that appear in the scoping unit and there shall be no other IMPLICIT statements
 7 in the scoping unit.

8 C580 (R561) If the minus and second *letter* appear, the second letter shall follow the first letter
 9 alphabetically.

10 A *letter-spec* consisting of two *letters* separated by a minus is equivalent to writing a list containing all
 11 of the letters in alphabetical order in the alphabetic sequence from the first letter through the second
 12 letter. For example, A–C is equivalent to A, B, C. The same letter shall not appear as a single letter, or
 13 be included in a range of letters, more than once in all of the IMPLICIT statements in a scoping unit.

14 In each scoping unit, there is a mapping, which may be null, between each of the letters A, B, ..., Z
 15 and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in
 16 its *letter-spec-list*. IMPLICIT NONE specifies the null mapping for all the letters. If a mapping is not
 17 specified for a letter, the default for a program unit or an interface body is default integer if the letter
 18 is I, J, ..., or N and default real otherwise, and the default for an internal or module procedure is the
 19 mapping in the host scoping unit.

20 Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic
 21 function, and is not made accessible by use association or host association is declared implicitly to be of
 22 the type (and type parameters) mapped from the first letter of its name, provided the mapping is not
 23 null. The mapping for the first letter of the data entity shall either have been established by a prior
 24 IMPLICIT statement or be the default mapping for the letter. The mapping may be to a derived type
 25 that is inaccessible in the local scope if the derived type is accessible in the host scoping unit. The data
 26 entity is treated as if it were declared in an explicit type declaration in the outermost scoping unit in
 27 which it appears. An explicit type specification in a FUNCTION statement overrides an IMPLICIT
 28 statement for the name of the result variable of that function subprogram.

NOTE 5.36

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)

```

NOTE 5.36 (cont.)

```

C = (3.0, 2.0)      ! C is implicitly declared COMPLEX
...
CONTAINS
  SUBROUTINE SUB1
    IMPLICIT INTEGER (A, C)
    C = (0.0, 0.0)   ! C is host associated and of
                    ! type complex
    Z = 1.0          ! Z is implicitly declared REAL
    A = 2            ! A is implicitly declared INTEGER
    CC = 1           ! CC is implicitly declared INTEGER
    ...
  END SUBROUTINE SUB1
  SUBROUTINE SUB2
    Z = 2.0          ! Z is implicitly declared REAL and
                    ! is different from the variable of
                    ! the same name in SUB1
    ...
  END SUBROUTINE SUB2
  SUBROUTINE SUB3
    USE EXAMPLE_MODULE ! Accesses integer function FUN
                      ! by use association
    Q = FUN (K)        ! Q is implicitly declared REAL and
    ...               ! K is implicitly declared INTEGER
  END SUBROUTINE SUB3
END SUBROUTINE SUB

```

NOTE 5.37

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.

1 5.6 NAMELIST statement

1 A **NAMELIST statement** specifies a group of named data objects, which may be referred to by a
 2 single name for the purpose of data transfer (9.5, 10.11).

3 R562 *namelist-stmt* is **NAMELIST** ■
 4 ■ / *namelist-group-name* / *namelist-group-object-list* ■
 5 ■ [[,] / *namelist-group-name* / ■
 6 ■ *namelist-group-object-list*] ...

7 C581 (R562) The *namelist-group-name* shall not be a name accessed by use association.

8 R563 *namelist-group-object* is *variable-name*

9 C582 (R563) A *namelist-group-object* shall not be an assumed-size array.

10 C583 (R562) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-*
 11 *name* has the PUBLIC attribute.

12 The order in which the variables are specified in the **NAMELIST** statement determines the order in
 13 which the values appear on output.

14 Any *namelist-group-name* may occur more than once in the **NAMELIST** statements in a scoping unit.
 15 The *namelist-group-object-list* following each successive appearance of the same *namelist-group-name* in
 16 a scoping unit is treated as a continuation of the list for that *namelist-group-name*.

17 A namelist group object may be a member of more than one namelist group.

18 A namelist group object shall either be accessed by use or host association or shall have its type, type
 19 parameters, and shape specified by previous specification statements or the procedure heading in the
 20 same scoping unit or by the implicit typing rules in effect for the scoping unit. If a namelist group object
 21 is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall
 22 confirm the implied type and type parameters.

NOTE 5.38

An example of a **NAMELIST** statement is:

```
NAMELIST /NLIST/ A, B, C
```

23 5.7 Storage association of data objects

24 5.7.1 EQUIVALENCE statement

25 5.7.1.1 General

26 An **EQUIVALENCE statement** is used to specify the sharing of storage units by two or more objects
 27 in a scoping unit. This causes storage association (16.5.3) of the objects that share the storage units.

NOTE 5.39

The co-size of a co-array is not a constant, therefore co-arrays are not allowed in **COMMON** or **EQUIVALENCE**.

28 If the equivalenced objects have differing type or type parameters, the **EQUIVALENCE** statement does
 29 not cause type conversion or imply mathematical equivalence. If a scalar and an array are equivalenced,
 30 the scalar does not have array properties and the array does not have the properties of a scalar.

R564 *equivalence-stmt* is **EQUIVALENCE** *equivalence-set-list*

- 1
- 2 R565 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)
- 3 R566 *equivalence-object* **is** *variable-name*
- 4 **or** *array-element*
- 5 **or** *substring*
- 6 C584 (R566) An *equivalence-object* shall not be a designator with a base object that is a dummy
- 7 argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ultimate
- 8 component, an object of a nonsequence derived type, an object of a derived type that has
- 9 a pointer at any level of component selection, an automatic object, a function name, an entry
- 10 name, a result name, a variable with the BIND attribute, a variable in a common block that
- 11 has the BIND attribute, or a named constant.
- 12 C585 (R566) An *equivalence-object* shall not be a designator that has more than one *part-ref*.
- 13 C586 (R566) An *equivalence-object* shall not be a co-array or a subobject thereof.
- 14 C587 (R566) An *equivalence-object* shall not have the TARGET attribute.
- 15 C588 (R566) Each subscript or substring range expression in an *equivalence-object* shall be an integer
- 16 initialization expression (7.1.12).
- 17 C589 (R565) If an *equivalence-object* is of type default integer, default real, double precision real,
- 18 default complex, default logical, default bits, or numeric sequence type, all of the objects in the
- 19 equivalence set shall be of these types.
- 20 C590 (R565) If an *equivalence-object* is of type default character or character sequence type, all of the
- 21 objects in the equivalence set shall be of these types.
- 22 C591 (R565) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or
- 23 character sequence type, all of the objects in the equivalence set shall be of the same type with
- 24 the same type parameter values.
- 25 C592 (R565) If an *equivalence-object* is of an intrinsic type other than default integer, default real,
- 26 double precision real, default complex, default logical, or default character, all of the objects in
- 27 the equivalence set shall be of the same type with the same kind type parameter value.
- 28 C593 (R566) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equiv-
- 29 alence set shall have the PROTECTED attribute.
- 30 C594 (R566) The name of an *equivalence-object* shall not be a name made accessible by use association.
- 31 C595 (R566) A *substring* shall not have length zero.

NOTE 5.40

The EQUIVALENCE statement allows the equivalencing of sequence structures and the equivalencing of objects of intrinsic type with nondefault type parameters, but there are strict rules regarding the appearance of these objects in an EQUIVALENCE statement.

A structure that appears in an EQUIVALENCE statement shall be a sequence structure. If a sequence structure is not of numeric sequence type or of character sequence type, it shall be equivalenced only to objects of the same type with the same type parameter values.

A structure of a numeric sequence type shall be equivalenced only to another structure of a numeric sequence type, an object of default integer type, default real type, double precision real type, default complex type, default logical type, or default bits such that components of the structure ultimately

NOTE 5.40 (cont.)

become associated only with objects of these types.

A structure of a character sequence type shall be equivalenced only to an object of default character type or another structure of a character sequence type.

An object of intrinsic type with nondefault kind type parameters shall not be equivalenced to objects of different type or kind type parameters.

Further rules on the interaction of EQUIVALENCE statements and default initialization are given in 16.5.3.4.

1 **5.7.1.2 Equivalence association**

2 An EQUIVALENCE statement specifies that the storage sequences (16.5.3.2) of the data objects specified
 3 in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if
 4 any, have the same first storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any,
 5 are storage associated with one another and with the first storage unit of any nonzero-sized sequences.
 6 This causes the storage association of the data objects in the *equivalence-set* and may cause storage
 7 association of other data objects.

8 **5.7.1.3 Equivalence of default character objects**

9 A data object of type default character shall not be equivalenced to an object that is not of type default
 10 character and not of a character sequence type. The lengths of equivalenced default character objects
 11 need not be the same.

12 An EQUIVALENCE statement specifies that the storage sequences of all the default character data
 13 objects specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the
 14 *equivalence-set*, if any, have the same first character storage unit, and all of the zero-sized sequences in
 15 the *equivalence-set*, if any, are storage associated with one another and with the first character storage
 16 unit of any nonzero-sized sequences. This causes the storage association of the data objects in the
 17 *equivalence-set* and may cause storage association of other data objects.

NOTE 5.41

For example, using the declarations:

```
CHARACTER (LEN = 4) :: A, B
CHARACTER (LEN = 3) :: C (2)
EQUIVALENCE (A, C (1)), (B, C (2))
```

the association of A, B, and C can be illustrated graphically as:

1	2	3	4	5	6	7
---	---	A	---			
			---	---	B	---
---	C(1)	---	---	C(2)	---	

18 **5.7.1.4 Array names and array element designators**

19 For a nonzero-sized array, the use of the array name unqualified by a subscript list as an *equivalence-*
 20 *object* has the same effect as using an array element designator that identifies the first element of the
 21 array.

1 5.7.1.5 Restrictions on EQUIVALENCE statements

- 2 An EQUIVALENCE statement shall not specify that the same storage unit is to occur more than once
3 in a storage sequence.

NOTE 5.42

For example:

```
REAL, DIMENSION (2) :: A
REAL :: B
EQUIVALENCE (A (1), B), (A (2), B) ! Not standard-conforming
```

is prohibited, because it would specify the same storage unit for A (1) and A (2).

- 4 An EQUIVALENCE statement shall not specify that consecutive storage units are to be nonconsecutive.

NOTE 5.43

For example, the following is prohibited:

```
REAL A (2)
DOUBLE PRECISION D (2)
EQUIVALENCE (A (1), D (1)), (A (2), D (2)) ! Not standard-conforming
```

5 5.7.2 COMMON statement

6 5.7.2.1 General

- 7 The **COMMON statement** specifies blocks of physical storage, called **common blocks**, that can be
8 accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global
9 data facility based on storage association (16.5.3).

- 10 The common blocks specified by the COMMON statement may be named and are called **named com-**
11 **mon blocks**, or may be unnamed and are called **blank common**.

12 R567 *common-stmt* **is** **COMMON** ■
13 ■ [/ [*common-block-name*] /] *common-block-object-list* ■
14 ■ [[,] / [*common-block-name*] / ■
15 ■ *common-block-object-list*] ...

16 R568 *common-block-object* **is** *variable-name* [(*array-spec*)]
17 **or** *proc-pointer-name*

18 C596 (R568) An *array-spec* in a *common-block-object* shall be an *explicit-shape-spec-list*.

19 C597 (R568) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all
20 *common-block-object-lists* within a scoping unit.

21 C598 (R568) A *common-block-object* shall not be a dummy argument, an allocatable variable, a
22 derived-type object with an ultimate component that is allocatable, an automatic object, a
23 function name, an entry name, a variable with the BIND attribute, a co-array, or a result name.

24 C599 (R568) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.2.3) or a
25 type with the BIND attribute and it shall have no default initialization.

26 C5100 (R568) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use

1 association.

2 In each COMMON statement, the data objects whose names appear in a common block object list
 3 following a common block name are declared to be in that common block. If the first common block
 4 name is omitted, all data objects whose names appear in the first common block object list are specified to
 5 be in blank common. Alternatively, the appearance of two slashes with no common block name between
 6 them declares the data objects whose names appear in the common block object list that follows to be
 7 in blank common.

8 Any common block name or an omitted common block name for blank common may occur more than
 9 once in one or more COMMON statements in a scoping unit. The common block list following each
 10 successive appearance of the same common block name in a scoping unit is treated as a continuation of
 11 the list for that common block name. Similarly, each blank common block object list in a scoping unit
 12 is treated as a continuation of blank common.

13 The form *variable-name (array-spec)* specifies the DIMENSION attribute for that variable.

14 If derived-type objects of numeric sequence type (4.5.2) or character sequence type (4.5.2) appear in
 15 common, it is as if the individual components were enumerated directly in the common list.

NOTE 5.44

Examples of COMMON statements are:

```
COMMON /BLOCKA/ A, B, D (10, 30)
COMMON I, J, K
```

16 5.7.2.2 Common block storage sequence

17 For each common block in a scoping unit, a **common block storage sequence** is formed as follows:

- 18 (1) A storage sequence is formed consisting of the sequence of storage units in the storage
 19 sequences (16.5.3.2) of all data objects in the common block object lists for the common
 20 block. The order of the storage sequences is the same as the order of the appearance of the
 21 common block object lists in the scoping unit.
- 22 (2) The storage sequence formed in (1) is extended to include all storage units of any storage
 23 sequence associated with it by equivalence association. The sequence shall be extended only
 24 by adding storage units beyond the last storage unit. Data objects associated with an entity
 25 in a common block are considered to be in that common block.

26 Only COMMON statements and EQUIVALENCE statements appearing in the scoping unit contribute
 27 to common block storage sequences formed in that scoping unit.

28 5.7.2.3 Size of a common block

29 The **size of a common block** is the size of its common block storage sequence, including any extensions
 30 of the sequence resulting from equivalence association.

31 5.7.2.4 Common association

32 Within a program, the common block storage sequences of all nonzero-sized common blocks with the
 33 same name have the same first storage unit, and the common block storage sequences of all zero-sized
 34 common blocks with the same name are storage associated with one another. Within a program, the
 35 common block storage sequences of all nonzero-sized blank common blocks have the same first storage
 36 unit and the storage sequences of all zero-sized blank common blocks are associated with one another and
 37 with the first storage unit of any nonzero-sized blank common blocks. This results in the association of

- 1 objects in different scoping units. Use association or host association may cause these associated objects
2 to be accessible in the same scoping unit.
- 3 A nonpointer object of default integer type, default real type, double precision real type, default complex
4 type, default logical type, or numeric sequence type shall be associated only with nonpointer objects of
5 these types.
- 6 A nonpointer object of type default character or character sequence type shall be associated only with
7 nonpointer objects of these types.
- 8 A nonpointer object of a derived type that is not a numeric sequence or character sequence type shall
9 be associated only with nonpointer objects of the same type with the same type parameter values.
- 10 A nonpointer object of intrinsic type other than default integer, default real, double precision real,
11 default complex, default logical, or default character shall be associated only with nonpointer objects of
12 the same type and type parameters.
- 13 A data pointer shall be storage associated only with data pointers of the same type and rank. Data
14 pointers that are storage associated shall have deferred the same type parameters; corresponding non-
15 deferred type parameters shall have the same value. A procedure pointer shall be storage associated
16 only with another procedure pointer; either both interfaces shall be explicit or both interfaces shall be
17 implicit. If the interfaces are explicit, the characteristics shall be the same. If the interfaces are implicit,
18 either both shall be subroutines or both shall be functions with the same type and type parameters.
- 19 An object with the TARGET attribute shall be storage associated only with another object that has
20 the TARGET attribute and the same type and type parameters.

NOTE 5.45

A common block is permitted to contain sequences of different storage units, provided each scoping unit that accesses the common block specifies an identical sequence of storage units for the common block. For example, this allows a single common block to contain both numeric and character storage units.

Association in different scoping units between objects of default type, objects of double precision real type, and sequence structures is permitted according to the rules for equivalence objects (5.7.1).

21 5.7.2.5 Differences between named common and blank common

22 A blank common block has the same properties as a named common block, except for the following.

- 23 (1) Execution of a RETURN or END statement might cause data objects in a named common
24 block to become undefined unless the common block has the SAVE attribute or is declared
25 in a module or submodule, but never causes data objects in blank common to become
26 undefined (16.6.6).
- 27 (2) Named common blocks of the same name shall be of the same size in all scoping units of a
28 program in which they appear, but blank common blocks may be of different sizes.
- 29 (3) A data object in a named common block may be initially defined by means of a DATA
30 statement or type declaration statement in a block data program unit (11.3), but objects in
31 blank common shall not be initially defined.

32 5.7.3 Restrictions on common and equivalence

33 An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to
34 be associated.

- 1 Equivalence association shall not cause a derived-type object with default initialization to be associated
- 2 with an object in a common block.
- 3 Equivalence association shall not cause a common block storage sequence to be extended by adding
- 4 storage units preceding the first storage unit of the first object specified in a COMMON statement for
- 5 the common block.

NOTE 5.46

For example, the following is not permitted:

```
COMMON /X/ A  
REAL B (2)  
EQUIVALENCE (A, B (2))      ! Not standard-conforming
```

6 Use of data objects

The appearance of a data object designator in a context that requires its value is termed a reference. A reference is permitted only if the data object is defined. A reference to a pointer is permitted only if the pointer is associated with a target object that is defined. A data object becomes defined with a value when events described in 16.6.5 occur.

R601 *variable* **is** *designator*
or *expr*

C601 (R601) *designator* shall not be a constant or a subobject of a constant.

C602 (R601) *expr* shall be a reference to a function that has a pointer result.

A variable is either the data object denoted by *designator* or the target of *expr*.

R602 *variable-name* **is** *name*

C603 (R602) *variable-name* shall be the name of a variable.

R603 *designator* **is** *object-name*
or *array-element*
or *array-section*
or *complex-part-designator*
or *structure-component*
or *substring*

R604 *logical-variable* **is** *variable*

C604 (R604) *logical-variable* shall be of type logical.

R605 *default-logical-variable* **is** *variable*

C605 (R605) *default-logical-variable* shall be of type default logical.

R606 *char-variable* **is** *variable*

C606 (R606) *char-variable* shall be of type character.

R607 *default-char-variable* **is** *variable*

C607 (R607) *default-char-variable* shall be of type default character.

R608 *int-variable* **is** *variable*

C608 (R608) *int-variable* shall be of type integer.

NOTE 6.1

For example, given the declarations:

```
CHARACTER (10)  A, B (10)
TYPE (PERSON)  P    ! See Note 4.20
```

then A, B, B (1), B (1:5), P % AGE, and A (1:1) are all variables.

1 A constant (3.2.2) is a literal constant or a named constant. A literal constant is a scalar denoted by a
 2 syntactic form, which indicates its type, type parameters, and value. A named constant is a constant
 3 that has a name; the name has the PARAMETER attribute (5.3.12, 5.4.10). A reference to a constant
 4 is always permitted; redefinition of a constant is never permitted.

5 6.1 Scalars

6 6.1.1 Substrings

7 A **substring** is a contiguous portion of a character string (4.4.5).

8 R609 *substring* **is** *parent-string* (*substring-range*)

9 R610 *parent-string* **is** *scalar-variable-name*

10 **or** *array-element*

11 **or** *scalar-structure-component*

12 **or** *scalar-constant*

13 R611 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

14 C609 (R610) *parent-string* shall be of type character.

15 The value of the first *scalar-int-expr* in *substring-range* is called the **starting point** and the value of
 16 the second one is called the **ending point**. The length of a substring is the number of characters in the
 17 substring and is $\text{MAX}(l - f + 1, 0)$, where f and l are the starting and ending points, respectively.

18 Let the characters in the parent string be numbered 1, 2, 3, ..., n , where n is the length of the parent
 19 string. Then the characters in the substring are those from the parent string from the starting point and
 20 proceeding in sequence up to and including the ending point. Both the starting point and the ending
 21 point shall be within the range 1, 2, ..., n unless the starting point exceeds the ending point, in which
 22 case the substring has length zero. If the starting point is not specified, the default value is 1. If the
 23 ending point is not specified, the default value is n .

24 If the parent is a variable, the substring is also a variable.

NOTE 6.2

Examples of character substrings are:

B(1)(1:5)	array element as parent string
P%NAME(1:1)	structure component as parent string
ID(4:9)	scalar variable name as parent string
'0123456789'(N:N)	character constant as parent string

25 6.1.2 Structure components

26 A **structure component** is part of an object of derived type; it may be referenced by an object
 27 designator. A structure component may be a scalar or an array.

28 R612 *data-ref* **is** *part-ref* [% *part-ref*] ...

29 R613 *part-ref* **is** *part-name* [(*section-subscript-list*)] [*image-selector*]

30 C610 (R612) Each *part-name* except the rightmost shall be of derived type.

31 C611 (R612) Each *part-name* except the leftmost shall be the name of a component of the declared

- 1 type of the preceding *part-name*.
- 2 C612 (R612) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.
- 3 C613 (R612) The leftmost *part-name* shall be the name of a data object.
- 4 C614 (R613) If a *section-subscript-list* appears, the number of *section-subscripts* shall equal the rank
5 of *part-name*.
- 6 C615 (R613) If *image-selector* appears, the number of *co-subscripts* shall be equal to the co-rank of
7 *part-name*.
- 8 C616 (R613) If *image-selector* appears and *part-name* is an array, *section-subscript-list* shall appear.
- 9 C617 (R612) If *image-selector* appears, *data-ref* shall not be, or have a direct component, of type
10 IMAGE_TEAM (13.8.2.7), C_PTR, or C_FUNPTR (15.3.3).
- 11 C618 (R612) A *data-ref* that is a co-indexed object shall not have a pointer ultimate component.

NOTE 6.3

This restriction is needed to avoid a disallowed pointer assignment for a component, such as

Z[P] = Z ! Not allowed if Z has a pointer component
Z = Z[P] ! Not allowed if Z has a pointer component

- 12 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
13 a section subscript list is the number of subscript triplets and vector subscripts in the list.
- 14 C619 (R612) There shall not be more than one *part-ref* with nonzero rank. A *part-name* to the right
15 of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.
- 16 The rank of a *data-ref* is the rank of the *part-ref* with nonzero rank, if any; otherwise, the rank is zero.
- 17 The **base object** of a *data-ref* is the data object whose name is the leftmost part name.
- 18 The type and type parameters, if any, of a *data-ref* are those of the rightmost part name.
- 19 A *data-ref* with more than one *part-ref* is a subobject of its base object if none of the *part-names*,
20 except for possibly the rightmost, are pointers. If the rightmost *part-name* is the only pointer, then the
21 *data-ref* is a subobject of its base object in contexts that pertain to its pointer association status but
22 not in any other contexts.

NOTE 6.4

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.

- 23 R614 *structure-component* **is** *data-ref*
- 24 C620 (R614) There shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form
25 *part-name*.
- 26 A structure component shall be neither referenced nor defined before the declaration of the base object.
- 27 A structure component is a pointer only if the rightmost part name is defined to have the POINTER
28 attribute.

NOTE 6.5

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.3.1.

NOTE 6.6

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.

- 1 A **subcomponent** of an object of derived type is a component of that object or of a subobject of that
- 2 object.
- 3 A *data-ref* shall not be a co-indexed object that has a pointer subcomponent. A *data-ref* that is a
- 4 co-indexed object shall not be, or have a subcomponent, of type IMAGE_TEAM (13.8.2.7), C_PTR, or
- 5 C_FUNPTR (15.3.3).

J3 internal note**Unresolved Technical Issue 089**

I'm just so sure the sky is going to fall in if the user dares to do this, but why are we adding uncheckable requirements? (Well, ok it's theoretically checkable at runtime only.)

And why is this requirement placed here? None of the other stuff which uses the term "sub-component" is here, this is just the definition. It probably belongs in a separate subclause for co-indexed objects... which might clean up some of the other mess that is being made of clause 6.

We *REALLY* ought not to be placing type requirements on runtime values. That is *UNACCEPTABLE*. I am not kidding – note the careful design of the object-oriented stuff to avoid such requirements. If the co-array folk don't have the time to design the facility properly, omit it, don't go and make a complete pig's ear of the standard.

It is not beyond the wit of man to design this to be compile-time checkable.

Back to the original question – why is the sky going to fall in? I do not believe that the consequences of allowing this justify the complexity of these arbitrary restrictions. *Furthermore*, the restrictions are ineffective. There are standard-conforming ways (using TRANSFER) to "hide" the type of the value and restore it later, on another image.

6.1.3 Complex parts

- 7 A **complex part designator** is used to designate the real or imaginary part of a complex data object,
- 8 independently of the other part.

9 R615 *complex-part-designator* **is** *designator* % RE
 10 **or** *designator* % IM

11 C621 (R615) The *designator* shall be of complex type.

- 12 If *complex-part-designator* is *designator*%RE it designates the real part of *designator*. If it is *designa-*
- 13 *tor*%IM it designates the imaginary part of *designator*. The type of a *complex-part-designator* is real,

1 and its kind and shape are those of the *designator*.

NOTE 6.7

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
```

2 6.1.4 Type parameter inquiry

3 A **type parameter inquiry** is used to inquire about a type parameter of a data object. It applies to
4 both intrinsic and derived types.

5 R616 *type-param-inquiry* **is** *designator % type-param-name*

6 C622 (R616) The *type-param-name* shall be the name of a type parameter of the declared type of the
7 object designated by the *designator*.

8 A deferred type parameter of a pointer that is not associated or of an unallocated allocatable variable
9 shall not be inquired about.

NOTE 6.8

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 may 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 6.9

The following are examples of type parameter inquiries:

```
a%kind      !-- A is real. Same value as KIND(a).
s%len       !-- S is character. Same value as LEN(s).
b(10)%kind  !-- Inquiry about an array element.
p%dim       !-- P is of the derived type general_point.
```

See Note 4.27 for the definition of the `general_point` type used in the last example above.

10 6.2 Arrays

11 An **array** is a set of scalar data, all of the same type and type parameters, whose individual elements
12 are arranged in a rectangular pattern. The scalar data that make up an array are the **array elements**.

13 No order of reference to the elements of an array is indicated by the appearance of the array designator,
14 except where array element ordering (6.2.2.1) is specified.

15 6.2.1 Whole arrays

16 A **whole array** is a named array, which may be either a named constant (5.3.12, 5.4.10) or a variable;
17 no subscript list is appended to the name.

1 The appearance of a whole array variable in an executable construct specifies all the elements of the
 2 array (2.4.5). An assumed-size array is permitted to appear as a whole array in an executable construct
 3 only as an actual argument in a procedure reference that does not require the shape.

4 The appearance of a whole array name in a nonexecutable statement specifies the entire array except
 5 for the appearance of a whole array name in an equivalence set (5.7.1.4).

6 6.2.2 Array elements and array sections

7 R617 *array-element* **is** *data-ref*

8 C623 (R617) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

9 R618 *array-section* **is** *data-ref* [(*substring-range*)]
 10 **or** *complex-part-designator*

11 C624 (R618) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have
 12 a *section-subscript-list* with nonzero rank, another *part-ref* shall have nonzero rank, or the
 13 *complex-part-designator* shall be an array.

14 C625 (R618) If a *substring-range* appears, the rightmost *part-name* shall be of type character.

15 R619 *subscript* **is** *scalar-int-expr*

16 R620 *section-subscript* **is** *subscript*
 17 **or** *subscript-triplet*
 18 **or** *vector-subscript*

19 R621 *subscript-triplet* **is** [*subscript*] : [*subscript*] [: *stride*]

20 R622 *stride* **is** *scalar-int-expr*

21 R623 *vector-subscript* **is** *int-expr*

22 C626 (R623) A *vector-subscript* shall be an integer array expression of rank one.

23 C627 (R621) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension
 24 of an assumed-size array.

25 An array element is a scalar. An array section is an array. If a *substring-range* is present in an *array-*
 26 *section*, each element is the designated substring of the corresponding element of the array section.

27 The value of a subscript in an array element shall be within the bounds for that dimension.

NOTE 6.10

For example, with the declarations:

```
REAL A (10, 10)
CHARACTER (LEN = 10) B (5, 5, 5)
```

A (1, 2) is an array element, A (1:N:2, M) is a rank-one array section, and B (:, :, :) (2:3) is an array of shape (5, 5, 5) whose elements are substrings of length 2 of the corresponding elements of B.

NOTE 6.11

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

NOTE 6.11 (cont.)

ALLOCATABLE attribute.

NOTE 6.12

Examples of array elements and array sections are:

ARRAY_A(1:N:2)%ARRAY_B(I, J)%STRING(K)(:)	array section
SCALAR_PARENT%ARRAY_FIELD(J)	array element
SCALAR_PARENT%ARRAY_FIELD(1:N)	array section
SCALAR_PARENT%ARRAY_FIELD(1:N)%SCALAR_FIELD	array section

6.2.2.1 Array element order

- The elements of an array form a sequence known as the **array element order**. The position of an array element in this sequence is determined by the subscript order value of the subscript list designating the element. The subscript order value is computed from the formulas in Table 6.1.

Table 6.1: **Subscript order value**

Rank	Subscript bounds	Subscript list	Subscript order value
1	$j_1:k_1$	s_1	$1 + (s_1 - j_1)$
2	$j_1:k_1, j_2:k_2$	s_1, s_2	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$
3	$j_1:k_1, j_2:k_2, j_3:k_3$	s_1, s_2, s_3	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$
.	.	.	.
.	.	.	.
.	.	.	.
15	$j_1:k_1, \dots, j_{15}:k_{15}$	$s_1, \dots, s_{15}$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$ $+ \dots$ $+ (s_{15} - j_{15}) \times d_{14}$ $\times d_{13} \times \dots \times d_1$
Notes for Table 6.1:			
1) $d_i = \max(k_i - j_i + 1, 0)$ is the size of the i th dimension.			
2) If the size of the array is nonzero, $j_i \leq s_i \leq k_i$ for all $i = 1, 2, \dots, 15$.			

6.2.2.2 Array sections

- An **array section** is an array subobject optionally followed by a substring range.
- 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.
- 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

1 element is the number of integer values in the sequence indicated by the *i*th subscript triplet or vector
 2 subscript. If any of these sequences is empty, the array section has size zero. The subscript order of the
 3 elements of an array section is that of the array data object that the array section represents.

4 6.2.2.2.1 Subscript triplet

5 A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values.
 6 The third expression in the subscript triplet is the increment between the subscript values and is called
 7 the **stride**. The subscripts and stride of a subscript triplet are optional. An omitted first subscript in a
 8 subscript triplet is equivalent to a subscript whose value is the lower bound for the array and an omitted
 9 second subscript is equivalent to the upper bound. An omitted stride is equivalent to a stride of 1.

10 The stride shall not be zero.

11 When the stride is positive, the subscripts specified by a triplet form a regularly spaced sequence of
 12 integers beginning with the first subscript and proceeding in increments of the stride to the largest such
 13 integer not greater than the second subscript; the sequence is empty if the first subscript is greater than
 14 the second.

NOTE 6.13

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)

15 When the stride is negative, the sequence begins with the first subscript and proceeds in increments of
 16 the stride down to the smallest such integer equal to or greater than the second subscript; the sequence
 17 is empty if the second subscript is greater than the first.

NOTE 6.14

For example, if an array is declared B (10), the section B (9 : 1 : -2) is the array of shape (5) whose elements are B (9), B (7), B (5), B (3), and B (1), in that order.

NOTE 6.15

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.

18 6.2.2.2.2 Vector subscript

19 A **vector subscript** designates a sequence of subscripts corresponding to the values of the elements
 20 of the expression. Each element of the expression shall be defined. A **many-one array section** is an
 21 array section with a vector subscript having two or more elements with the same value. A many-one
 22 array section shall appear neither on the left of the equals in an assignment statement nor as an input
 23 item in a READ statement.

24 An array section with a vector subscript shall not be argument associated with a dummy array that
 25 is defined or redefined. An array section with a vector subscript shall not be the target in a pointer
 26 assignment statement. An array section with a vector subscript shall not be an internal file.

NOTE 6.16

For example, suppose Z is a two-dimensional array of shape $[5, 7]$ and U and V are one-dimensional arrays of shape (3) and (4) , respectively. Assume the values of U and V are:

$U = [1, 3, 2]$
 $V = [2, 1, 1, 3]$

Then $Z(3, V)$ consists of elements from the third row of Z in the order:

$Z(3, 2) \quad Z(3, 1) \quad Z(3, 1) \quad Z(3, 3)$

and $Z(U, 2)$ consists of the column elements:

$Z(1, 2) \quad Z(3, 2) \quad Z(2, 2)$

and $Z(U, V)$ consists of the elements:

$Z(1, 2) \quad Z(1, 1) \quad Z(1, 1) \quad Z(1, 3)$
 $Z(3, 2) \quad Z(3, 1) \quad Z(3, 1) \quad Z(3, 3)$
 $Z(2, 2) \quad Z(2, 1) \quad Z(2, 1) \quad Z(2, 3)$

Because $Z(3, V)$ and $Z(U, V)$ contain duplicate elements from Z , the sections $Z(3, V)$ and $Z(U, V)$ shall not be redefined as sections.

1 **6.2.3 Image selectors**

2 An **image selector** specifies the image index for co-array data.

3 R624 *image-selector* **is** *lbracket co-subscript-list rbracket*

4 R625 *co-subscript* **is** *scalar-int-expr*

5 The number of co-subscripts shall be equal to the co-rank of the object. The value of a co-subscript in
 6 an image selector shall be within the co-bounds for its co-dimension. Taking account of the co-bounds,
 7 the co-subscript list in an image selector determines the image index in the same way that a subscript
 8 list in an array element determines the subscript order value (6.2.2.1), taking account of the bounds. An
 9 image selector shall specify an image index value that is not greater than the number of images.

NOTE 6.17

For example, if there are 16 images and the co-array 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.

10 **6.3 Dynamic association**

11 **6.3.1 ALLOCATE statement**

12 The **ALLOCATE statement** dynamically creates pointer targets and allocatable variables.

13 R626 *allocate-stmt* **is** `ALLOCATE ([type-spec ::] allocation-list ■
 ■ [, alloc-opt-list])`

1	R627	<i>alloc-opt</i>	is <i>ERRMSG</i> = <i>errmsg-variable</i>
2			or <i>MOLD</i> = <i>source-expr</i>
3			or <i>SOURCE</i> = <i>source-expr</i>
4			or <i>STAT</i> = <i>stat-variable</i>
5	R628	<i>stat-variable</i>	is <i>scalar-int-variable</i>
6	R629	<i>errmsg-variable</i>	is <i>scalar-default-char-variable</i>
7	R630	<i>source-expr</i>	is <i>expr</i>
8	R631	<i>allocation</i>	is <i>allocate-object</i> [(<i>allocate-shape-spec-list</i>)] ■
9			■ [<i>lbracket allocate-co-array-spec rbracket</i>]
10	R632	<i>allocate-object</i>	is <i>variable-name</i>
11			or <i>structure-component</i>
12	R633	<i>allocate-shape-spec</i>	is [<i>lower-bound-expr</i> :] <i>upper-bound-expr</i>
13	R634	<i>lower-bound-expr</i>	is <i>scalar-int-expr</i>
14	R635	<i>upper-bound-expr</i>	is <i>scalar-int-expr</i>
15	R636	<i>allocate-co-array-spec</i>	is [<i>allocate-co-shape-spec-list</i> ,] [<i>lower-bound-expr</i> :] *
16	R637	<i>allocate-co-shape-spec</i>	is [<i>lower-bound-expr</i> :] <i>upper-bound-expr</i>
17	C628	(R632) Each <i>allocate-object</i> shall be a nonprocedure pointer or an allocatable variable.	
18	C629	(R626) If any <i>allocate-object</i> has a deferred type parameter, is unlimited polymorphic, or is of	
19		abstract type, either <i>type-spec</i> or <i>source-expr</i> shall appear.	
20	C630	(R626) If a <i>type-spec</i> appears, it shall specify a type with which each <i>allocate-object</i> is type	
21		compatible.	
22	C631	(R626) A <i>type-param-value</i> in a <i>type-spec</i> shall be an asterisk if and only if each <i>allocate-object</i>	
23		is a dummy argument for which the corresponding type parameter is assumed.	
24	C632	(R626) If a <i>type-spec</i> appears, the kind type parameter values of each <i>allocate-object</i> shall be	
25		the same as the corresponding type parameter values of the <i>type-spec</i> .	
26	C633	(R631) If <i>allocate-object</i> is an array either <i>allocate-shape-spec-list</i> shall appear or <i>source-expr</i>	
27		shall appear and have the same rank as <i>allocate-object</i> . If <i>allocate-object</i> is scalar, <i>allocate-</i>	
28		<i>shape-spec-list</i> shall not appear.	
29	C634	(R631) An <i>allocate-co-array-spec</i> shall appear if and only if the <i>allocate-object</i> is a co-array.	
30	C635	(R631) The number of <i>allocate-shape-specs</i> in an <i>allocate-shape-spec-list</i> shall be the same as the	
31		rank of the <i>allocate-object</i> . The number of <i>allocate-co-shape-specs</i> in an <i>allocate-co-array-spec</i>	
32		shall be one less than the co-rank of the <i>allocate-object</i> .	
33	C636	(R627) No <i>alloc-opt</i> shall appear more than once in a given <i>alloc-opt-list</i> .	
34	C637	(R626) At most one of <i>source-expr</i> and <i>type-spec</i> shall appear.	
35	C638	(R626) Each <i>allocate-object</i> shall be type compatible (4.3.1.3) with <i>source-expr</i> . If <i>SOURCE</i> =	
36		appears, <i>source-expr</i> shall be a scalar or have the same rank as each <i>allocate-object</i> .	
37	C639	(R626) Corresponding kind type parameters of <i>allocate-object</i> and <i>source-expr</i> shall have the	

- 1 same values.
- 2 C640 (R626) *type-spec* shall not specify a type that has a co-array ultimate component.
- 3 C641 (R630) The declared type of *source-expr* shall not have a co-array ultimate component.
- 4 C642 (R632) An *allocate-object* shall not be a co-indexed object.

NOTE 6.18

If a co-array is of a derived type that has an allocatable component, the component shall be allocated by its own image:

```

TYPE(SOMETHING), ALLOCATABLE :: T[:]
...
ALLOCATE(T[*])           ! Allowed - implies synchronization
ALLOCATE(T%AAC(N))       ! Allowed - allocated by its own image
ALLOCATE(T[Q]%AAC(N))    ! Not allowed, because it is not
                        ! necessarily executed on image Q.

```

- 5 An *allocate-object* or a bound or type parameter of an *allocate-object* shall not depend on the value of
- 6 *stat-variable*, the value of *errmsg-variable*, or on the value, bounds, length type parameters, allocation
- 7 status, or association status of any *allocate-object* in the same ALLOCATE statement.
- 8 *source-expr* shall not be allocated within the ALLOCATE statement in which it appears; nor shall it
- 9 depend on the value, bounds, deferred type parameters, allocation status, or association status of any
- 10 *allocate-object* in that statement.
- 11 If *type-spec* is specified, each *allocate-object* is allocated with the specified dynamic type and type pa-
- 12 rameter values; if *source-expr* is specified, each *allocate-object* is allocated with the dynamic type and
- 13 type parameter values of *source-expr*; otherwise, each *allocate-object* is allocated with its dynamic type
- 14 the same as its declared type.
- 15 If *type-spec* appears and the value of a type parameter it specifies differs from the value of the corre-
- 16 sponding 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
- 18 of the corresponding type parameter of *source-expr*, an error condition occurs.
- 19 If a *type-param-value* in a *type-spec* in an ALLOCATE statement is an asterisk, it denotes the current
- 20 value of that assumed type parameter. If it is an expression, subsequent redefinition or undefinition of
- 21 any entity in the expression does not affect the type parameter value.

NOTE 6.19

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

- 22 When an ALLOCATE statement is executed for an array for which *allocate-shape-spec-list* is specified,
- 23 the values of the lower bound and upper bound expressions determine the bounds of the array. Subse-
- 24 quent redefinition or undefinition of any entities in the bound expressions do not affect the array bounds.
- 25 If the lower bound is omitted, the default value is 1. If the upper bound is less than the lower bound,
- 26 the extent in that dimension is zero and the array has zero size.
- 27 When an ALLOCATE statement is executed for a co-array, the values of the lower co-bound and upper
- 28 co-bound expressions determine the co-bounds of the co-array. Subsequent redefinition or undefinition

- 1 of any entities in the co-bound expressions do not affect the co-bounds. If the lower co-bound is omitted,
 2 the default value is 1. The upper co-bound shall not be less than the lower co-bound.
- 3 The value of each *lower-bound-expr* and each *upper-bound-expr* in an *allocate-co-array-spec* shall be the
 4 same on each image.
- 5 If an *allocation* specifies a co-array, its dynamic type and the values of each length type parameter shall
 6 be the same on each image.
- 7 There is implicit synchronization of all images in association with each ALLOCATE statement that
 8 involves one or more co-arrays. On each image, execution of the segment (8.5.1) following the statement
 9 is delayed until all other images have executed the same statement the same number of times.

NOTE 6.20

When an image executes an ALLOCATE statement, communication is not necessarily involved apart from any required for synchronization. The image allocates its co-array and records how the corresponding co-arrays on other images are to be addressed. The processor is not required to detect violations of the rule that the bounds are the same on all images, nor is it responsible for detecting or resolving deadlock problems (such as two images waiting on different ALLOCATE statements).

- 10 If *source-expr* is a pointer, it shall be associated with a target. If *source-expr* is allocatable, it shall be
 11 allocated.
- 12 When an ALLOCATE statement is executed for an array with no *allocate-shape-spec-list*, the bounds
 13 of *source-expr* determine the bounds of the array. Subsequent changes to the bounds of *source-expr* do
 14 not affect the array bounds.
- 15 If SOURCE= appears, *source-expr* shall be conformable (2.4.5) with *allocation*. If the value of a non-
 16 deferred length type parameter of *allocate-object* is different from the value of the corresponding type
 17 parameter of *source-expr*, an error condition occurs. On successful allocation, if *allocate-object* and
 18 *source-expr* have the same rank the value of *allocate-object* becomes that of *source-expr*, otherwise the
 19 value of each element of *allocate-object* becomes that of *source-expr*.
- 20 If MOLD= appears and *source-expr* is a variable, its value need not be defined.

J3 internal note**Unresolved Technical Issue 081**

Should we require allocatables to be allocated, or only when they have deferred type parameters? Furthermore, we should probably extend MOLD= (and SOURCE=) to be able to specify the shape, when no *array-spec* is provided, so the user can write

```
ALLOCATE(A,MOLD=B)
```

instead of

```
ALLOCATE(A(LBOUND(B,1):UBOUND(B,1),LBOUND(B,2):UBOUND(B,2)),MOLD=B)
```

NOTE 6.21

An example of an ALLOCATE statement in which the value and dynamic type are determined by reference to another object is:

```
CLASS(*), ALLOCATABLE :: NEW
CLASS(*), POINTER :: OLD
! ...
```

NOTE 6.21 (cont.)

```
ALLOCATE (NEW, SOURCE=OLD) ! Allocate NEW with the value and dynamic type of OLD
```

A more extensive example is given in C.3.2.

1 The STAT= specifier is described in 6.3.4.

2 If an error condition occurs during execution of an ALLOCATE statement that does not contain the
3 STAT= specifier, execution of the program is terminated.

4 The ERRMSG= specifier is described in 6.3.5.

5 **6.3.1.1 Allocation of allocatable variables**

6 The allocation status of an allocatable entity is one of the following at any time.

7 (1) The status of an allocatable variable becomes **allocated** if it is allocated by an ALLOCATE
8 statement, if it is allocated during assignment, or if it is given that status by the allocation
9 transfer procedure (13.7.124). An allocatable variable with this status may be referenced,
10 defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement.
11 The intrinsic function ALLOCATED (13.7.10) returns true for such a variable.

12 (2) An allocatable variable has a status of **unallocated** if it is not allocated. The status of
13 an allocatable variable becomes unallocated if it is deallocated (6.3.3) or if it is given that
14 status by the allocation transfer procedure. An allocatable variable with this status shall
15 not be referenced or defined. It shall not be supplied as an actual argument corresponding
16 to a nonallocatable dummy argument, except to certain intrinsic inquiry functions. It may
17 be allocated with the ALLOCATE statement. Deallocating it causes an error condition in
18 the DEALLOCATE statement. The intrinsic function ALLOCATED (13.7.10) returns false
19 for such a variable.

20 At the beginning of execution of a program, allocatable variables are unallocated.

21 When the allocation status of an allocatable variable changes, the allocation status of any associated
22 allocatable variable changes accordingly. Allocation of an allocatable variable establishes values for the
23 deferred type parameters of all associated allocatable variables.

24 An unsaved allocatable local variable of a procedure has a status of unallocated at the beginning of
25 each invocation of the procedure. An unsaved allocatable local variable of a module or submodule, or
26 a subobject thereof, has an initial status of unallocated. An unsaved local variable of a construct has a
27 status of unallocated at the beginning of each execution of the construct.

28 When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate
29 components have an allocation status of unallocated.

30 **6.3.1.2 Allocation of pointer targets**

31 Allocation of a pointer creates an object that implicitly has the TARGET attribute. Following successful
32 execution of an ALLOCATE statement for a pointer, the pointer is associated with the target and may
33 be used to reference or define the target. Additional pointers may become associated with the pointer
34 target or a part of the pointer target by pointer assignment. It is not an error to allocate a pointer
35 that is already associated with a target. In this case, a new pointer target is created as required by the
36 attributes of the pointer and any array bounds, type, and type parameters specified by the ALLOCATE
37 statement. The pointer is then associated with this new target. Any previous association of the pointer
38 with a target is broken. If the previous target had been created by allocation, it becomes inaccessible
39 unless other pointers are associated with it. The ASSOCIATED intrinsic function (13.7.15) may be used

- 1 to determine whether a pointer that does not have undefined association status is associated.
- 2 At the beginning of execution of a function whose result is a pointer, the association status of the result
- 3 pointer is undefined. Before such a function returns, it shall either associate a target with this pointer
- 4 or cause the association status of this pointer to become disassociated.

5 6.3.2 NULLIFY statement

6 The **NULLIFY statement** causes pointers to be disassociated.

7 R638 *nullify-stmt* **is** NULLIFY (*pointer-object-list*)

8 R639 *pointer-object* **is** *variable-name*

9 **or** *structure-component*

10 **or** *proc-pointer-name*

11 C643 (R639) Each *pointer-object* shall have the POINTER attribute.

12 A *pointer-object* shall not depend on the value, bounds, or association status of another *pointer-object*

13 in the same NULLIFY statement.

NOTE 6.22

When a NULLIFY statement is applied to a polymorphic pointer (4.3.1.3), its dynamic type becomes the declared type.

14 6.3.3 DEALLOCATE statement

15 The **DEALLOCATE statement** causes allocatable variables to be deallocated; it causes pointer tar-

16 gets to be deallocated and the pointers to be disassociated.

17 R640 *deallocate-stmt* **is** DEALLOCATE (*allocate-object-list* [, *dealloc-opt-list*])

18 C644 (R640) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

19 R641 *dealloc-opt* **is** STAT = *stat-variable*

20 **or** ERRMSG = *errmsg-variable*

21 C645 (R641) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

22 An *allocate-object* shall not depend on the value, bounds, allocation status, or association status of

23 another *allocate-object* in the same DEALLOCATE statement; it also shall not depend on the value of

24 the *stat-variable* or *errmsg-variable* in the same DEALLOCATE statement.

25 The STAT= specifier is described in 6.3.4.

26 If an error condition occurs during execution of a DEALLOCATE statement that does not contain the

27 STAT= specifier, execution of the program is terminated.

28 The ERRMSG= specifier is described in 6.3.5.

NOTE 6.23

An example of a DEALLOCATE statement is:

```
DEALLOCATE (X, B)
```

29 6.3.3.1 Deallocation of allocatable variables

- 1 Deallocating an unallocated allocatable variable causes an error condition in the DEALLOCATE state-
- 2 ment. Deallocating an allocatable variable with the TARGET attribute causes the pointer association
- 3 status of any pointer associated with it to become undefined.
- 4 When the execution of a procedure is terminated by execution of a RETURN or END statement, an
- 5 unsaved allocatable local variable of the procedure retains its allocation and definition status if it is a
- 6 function result variable or a subobject thereof; otherwise, it is deallocated.
- 7 When a BLOCK construct terminates, an unsaved allocatable local variable of the construct is deallo-
- 8 cated.

NOTE 6.24

The ALLOCATED intrinsic function may be used to determine whether a variable is allocated or unallocated.

- 9 If an executable construct references a function whose result is either allocatable or a structure with
- 10 a subobject that is allocatable, and the function reference is executed, an allocatable result and any
- 11 subobject that is an allocated allocatable entity in the result returned by the function is deallocated
- 12 after execution of the innermost executable construct containing the reference.
- 13 If a function whose result is either allocatable or a structure with an allocatable subobject is referenced
- 14 in the specification part of a scoping unit or BLOCK construct, and the function reference is executed,
- 15 an allocatable result and any subobject that is an allocated allocatable entity in the result returned by
- 16 the function is deallocated before execution of the executable constructs of the scoping unit or block.
- 17 When a procedure is invoked, any allocated allocatable object that is an actual argument associated with
- 18 an INTENT(OUT) allocatable dummy argument is deallocated; any allocated allocatable object that is
- 19 a subobject of an actual argument associated with an INTENT(OUT) dummy argument is deallocated.
- 20 When an intrinsic assignment statement (7.2.1.3) is executed, any non-co-array allocated allocatable
- 21 subobject of the variable is deallocated before the assignment takes place.
- 22 When a variable of derived type is deallocated, any allocated allocatable subobject is deallocated.
- 23 If an allocatable component is a subobject of a finalizable object, that object is finalized before the
- 24 component is automatically deallocated.
- 25 The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a
- 26 *dealloc-opt-list*.

NOTE 6.25

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.

- 27 There is implicit synchronization of all images in association with each DEALLOCATE statement that
- 28 involves one or more co-arrays. On each image, execution of the segment (8.5.1) following the statement

1 is delayed until all other images have executed the same statement the same number of times.

2 There is also an implicit synchronization of all images in association with the deallocation of a co-array
3 or co-array subcomponent caused by the execution of a RETURN or END statement or the termination
4 of a BLOCK construct.

5 **6.3.3.2 Deallocation of pointer targets**

6 If a pointer appears in a DEALLOCATE statement, its association status shall be defined. Deallocating
7 a pointer that is disassociated or whose target was not created by an ALLOCATE statement causes an
8 error condition in the DEALLOCATE statement. If a pointer is associated with an allocatable entity,
9 the pointer shall not be deallocated.

10 If a pointer appears in a DEALLOCATE statement, it shall be associated with the whole of an object
11 that was created by allocation. Deallocating a pointer target causes the pointer association status of
12 any other pointer that is associated with the target or a portion of the target to become undefined.

13 **6.3.4 STAT= specifier**

14 *stat-variable* shall not be allocated or deallocated within the ALLOCATE or DEALLOCATE statement
15 in which it appears; nor shall it depend on the value, bounds, deferred type parameters, allocation status,
16 or association status of any *allocate-object* in that statement.

17 If the STAT= specifier appears, successful execution of the ALLOCATE or DEALLOCATE statement
18 causes the *stat-variable* to become defined with a value of zero.

19 If an error condition occurs during the execution of the ALLOCATE or DEALLOCATE statement, the
20 *stat-variable* becomes defined with a processor-dependent positive integer value and each *allocate-object*
21 will have a processor-dependent status:

- 22 • each *allocate-object* that was successfully allocated shall have an allocation status of allocated or
23 a pointer association status of associated;
- 24 • each *allocate-object* that was successfully deallocated shall have an allocation status of unallocated
25 or a pointer association status of disassociated;
- 26 • each *allocate-object* that was not successfully allocated or deallocated shall retain its previous
27 allocation status or pointer association status.

NOTE 6.26

The status of objects that were not successfully allocated or deallocated can be individually checked with the ALLOCATED or ASSOCIATED intrinsic functions.

28 **6.3.5 ERRMSG= specifier**

29 *errmsg-variable* shall not be allocated or deallocated within the ALLOCATE or DEALLOCATE state-
30 ment in which it appears; nor shall it depend on the value, bounds, deferred type parameters, allocation
31 status, or association status of any *allocate-object* in that statement.

32 If an error condition occurs during execution of an ALLOCATE or DEALLOCATE statement, the
33 processor shall assign an explanatory message to *errmsg-variable*. If no such condition occurs, the
34 processor shall not change the value of *errmsg-variable*.

7 Expressions and assignment

7.1 Expressions

7.1.1 General

An **expression** represents either a data reference or a computation, and its value is either a scalar or an array. An expression is formed from operands, operators, and parentheses.

An operand is either a scalar or an array. An operation is either intrinsic (7.1.5) or defined (7.1.6). More complicated expressions can be formed using operands which are themselves expressions.

Evaluation of an expression produces a value, which has a type, type parameters (if appropriate), and a shape (7.1.9). The co-rank of an expression that is not a variable is zero.

7.1.2 Form of an expression

7.1.2.1 Expression categories

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.

These categories are related to the different operator precedence levels and, in general, are defined in terms of other categories. The simplest form of each expression category is a *primary*.

7.1.2.2 Primary

R701 *primary* is *constant*
 or *designator*
 or *array-constructor*
 or *structure-constructor*
 or *function-reference*
 or *type-param-inquiry*
 or *type-param-name*
 or (*expr*)

C701 (R701) The *type-param-name* shall be the name of a type parameter.

C702 (R701) The *designator* shall not be a whole assumed-size array.

NOTE 7.1

Examples of a *primary* are:

<u>Example</u>	<u>Syntactic class</u>
1.0	<i>constant</i>
'ABCDEFGHJKLMNOPQRSTUVWXYZ' (I:I)	<i>designator</i>
[1.0, 2.0]	<i>array-constructor</i>
PERSON (12, 'Jones')	<i>structure-constructor</i>
F (X, Y)	<i>function-reference</i>
X%KIND	<i>type-param-inquiry</i>

NOTE 7.1 (cont.)

KIND (S + T)	<i>type-param-name</i> (<i>expr</i>)

7.1.2.3 Level-1 expressions

Defined unary operators have the highest operator precedence (Table 7.2). Level-1 expressions are primaries optionally operated on by defined unary operators:

R702 *level-1-expr* **is** [*defined-unary-op*] *primary*

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

C703 (R703) A *defined-unary-op* shall not contain more than 63 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.2

Simple examples of a level-1 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>primary</i> (R701)
. INVERSE. B	<i>level-1-expr</i> (R702)

A more complicated example of a level-1 expression is:

. INVERSE. (A + B)

7.1.2.4 Level-2 expressions

Level-2 expressions are level-1 expressions optionally involving the numeric operators *power-op*, *mult-op*, and *add-op*.

R704 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R705 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R706 *level-2-expr* **is** [[*level-2-expr*] *add-op*] *add-operand*

R707 *power-op* **is** **

R708 *mult-op* **is** *

or /

R709 *add-op* **is** +

or -

NOTE 7.3

Simple examples of a level-2 expression are:

<u>Example</u>	<u>Syntactic class</u>	<u>Remarks</u>
A	<i>level-1-expr</i>	A is a <i>primary</i> . (R702)
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> . (R704)
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> . (R705)

NOTE 7.3 (cont.)

+1	<i>level-2-expr</i>	+ is an <i>add-op</i> and 1 is an <i>add-operand</i> . (R706)
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> . (R706)

A more complicated example of a level-2 expression is:

- A + D * E + B ** C

1 **7.1.2.5 Level-3 expressions**

2 Level-3 expressions are level-2 expressions optionally involving the character operator and bits concate-
3 nation operator *concat-op*.

4 R710 *level-3-expr* is [*level-3-expr concat-op*] *level-2-expr*

5 R711 *concat-op* is //

NOTE 7.4

Simple examples of a level-3 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-2-expr</i> (R706)
B // C	<i>level-3-expr</i> (R710)

A more complicated example of a level-3 expression is:

X // Y // 'ABCD'

6 **7.1.2.6 Level-4 expressions**

7 Level-4 expressions are level-3 expressions optionally involving the relational operators *rel-op*.

8 R712 *level-4-expr* is [*level-3-expr rel-op*] *level-3-expr*

9 R713 *rel-op* is .EQ.
10 or .NE.
11 or .LT.
12 or .LE.
13 or .GT.
14 or .GE.
15 or ==
16 or /=
17 or <
18 or <=
19 or >
20 or >=

NOTE 7.5

Simple examples of a level-4 expression are:

NOTE 7.5 (cont.)

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-3-expr</i> (R710)
B == C	<i>level-4-expr</i> (R712)
D < E	<i>level-4-expr</i> (R712)
A more complicated example of a level-4 expression is:	
(A + B) /= C	

7.1.2.7 Level-5 expressions

Level-5 expressions are level-4 expressions optionally involving the logical and bits operators *not-op*, *and-op*, *or-op*, and *equiv-op*.

R714	<i>and-operand</i>	is	[<i>not-op</i>] <i>level-4-expr</i>
R715	<i>or-operand</i>	is	[<i>or-operand and-op</i>] <i>and-operand</i>
R716	<i>equiv-operand</i>	is	[<i>equiv-operand or-op</i>] <i>or-operand</i>
R717	<i>level-5-expr</i>	is	[<i>level-5-expr equiv-op</i>] <i>equiv-operand</i>
R718	<i>not-op</i>	is	.NOT.
R719	<i>and-op</i>	is	.AND.
R720	<i>or-op</i>	is	.OR.
R721	<i>equiv-op</i>	is	.EQV.
		or	.NEQV.
		or	.XOR.

NOTE 7.6

Simple examples of a level-5 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-4-expr</i> (R712)
.NOT. B	<i>and-operand</i> (R714)
C .AND. D	<i>or-operand</i> (R715)
E .OR. F	<i>equiv-operand</i> (R716)
G .EQV. H	<i>level-5-expr</i> (R717)
S .NEQV. T	<i>level-5-expr</i> (R717)
A more complicated example of a level-5 expression is:	
A .AND. B .EQV. .NOT. C	

7.1.2.8 General form of an expression

Expressions are level-5 expressions optionally involving defined binary operators. Defined binary operators have the lowest operator precedence (Table 7.2).

R722	<i>expr</i>	is	[<i>expr defined-binary-op</i>] <i>level-5-expr</i>
R723	<i>defined-binary-op</i>	is	. letter [letter]

- 1 C704 (R723) A *defined-binary-op* shall not contain more than 63 letters and shall not be the same as
 2 any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.7

Simple examples of an expression are:

Example	Syntactic class
A	<i>level-5-expr</i> (R717)
B.UNION.C	<i>expr</i> (R722)

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)
```

7.1.3 Precedence of operators

- 4 There is a precedence among the intrinsic and extension operations corresponding to the form of expres-
 5 sions specified in 7.1.2, which determines the order in which the operands are combined unless the order
 6 is changed by the use of parentheses. This precedence order is summarized in Table 7.2.

Table 7.2: 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, Bits	.NOT.	.
Logical, Bits	.AND.	.
Logical, Bits	.OR.	.
Logical, Bits	.EQV., .NEQV., .XOR.	.
Extension	<i>defined-binary-op</i>	Lowest

- 7 The precedence of a defined operation is that of its operator.

NOTE 7.8

For example, in the expression

```
-A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (-); therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. The interpretation of the above expression is the same as the interpretation of the expression

```
- (A ** 2)
```

- 1 The general form of an expression (7.1.2) also establishes a precedence among operators in the same
- 2 syntactic class. This precedence determines the order in which the operands are to be combined in
- 3 determining the interpretation of the expression unless the order is changed by the use of parentheses.

NOTE 7.9

In interpreting a *level-2-expr* containing two or more binary operators + or −, each operand (*add-operand*) is combined from left to right. Similarly, the same left-to-right interpretation for a *mult-operand* in *add-operand*, as well as for other kinds of expressions, is a consequence of the general form. However, for interpreting a *mult-operand* expression when two or more exponentiation operators ** combine *level-1-expr* operands, each *level-1-expr* is combined from right to left.

For example, the expressions

```
2.1 + 3.4 + 4.9
2.1 * 3.4 * 4.9
2.1 / 3.4 / 4.9
2 ** 3 ** 4
'AB' // 'CD' // 'EF'
```

have the same interpretations as the expressions

```
(2.1 + 3.4) + 4.9
(2.1 * 3.4) * 4.9
(2.1 / 3.4) / 4.9
2 ** (3 ** 4)
('AB' // 'CD') // 'EF'
```

As a consequence of the general form (7.1.2), only the first *add-operand* of a *level-2-expr* may be preceded by the identity (+) or negation (−) operator. These formation rules do not permit expressions containing two consecutive numeric operators, such as A ** −B or A + −B. However, expressions such as A ** (−B) and A + (−B) are permitted. The rules do allow a binary operator or an intrinsic unary operator to be followed by a defined unary operator, such as:

```
A * .INVERSE. B
- .INVERSE. (B)
```

As another example, in the expression

```
A .OR. B .AND. C
```

the general form implies a higher precedence for the .AND. operator than for the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

```
A .OR. (B .AND. C)
```

NOTE 7.10

An expression may 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

NOTE 7.10 (cont.)

```
L .OR. ((A + B) >= C)
```

NOTE 7.11

If

- (1) The operator ****** is extended to type logical,
- (2) The operator **.STARSTAR.** is defined to duplicate the function of ****** on type real,
- (3) **.MINUS.** is defined to duplicate the unary operator **−**, and
- (4) 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 **7.1.4 Evaluation of operations**

- 2 An intrinsic operation requires the values of its operands.
- 3 The evaluation of a function reference shall neither affect nor be affected by the evaluation of any other
- 4 entity within the statement. If a function reference causes definition or undefinition of an actual argument
- 5 of the function, that argument or any associated entities shall not appear elsewhere in the same statement.
- 6 However, execution of a function reference in the logical expression in an IF statement (8.1.8.4), the mask
- 7 expression in a WHERE statement (7.2.3.1), or the subscripts and strides in a FORALL statement (7.2.4)
- 8 is permitted to define variables in the statement that is conditionally executed.

NOTE 7.12

For example, the statements

```
A (I) = F (I)
Y = G (X) + X
```

are prohibited if the reference to F defines or undefines I or the reference to G defines or undefines X.

However, in the statements

```
IF (F (X)) A = X
WHERE (G (X)) B = X
```

F or G may define X.

- 9 The appearance of an array constructor requires the evaluation of each *scalar-int-expr* of the *ac-implied-*
- 10 *do-control* in any *ac-implied-do* it may contain.
- 11 When an elemental binary operation is applied to a scalar and an array or to two arrays of the same
- 12 shape, the operation is performed element-by-element on corresponding array elements of the array
- 13 operands.

NOTE 7.13

For example, the array expression

```
A + B
```

NOTE 7.13 (cont.)

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.

- 1 When an elemental unary operator operates on an array operand, the operation is performed element-
 2 by-element, and the result is the same shape as the operand.

NOTE 7.14

If an elemental operation is intrinsically pure or is implemented by a pure elemental function (12.8), the element operations may be performed simultaneously or in any order.

3 **7.1.5 Intrinsic operations**

4 **7.1.5.1 Definitions**

5 An **intrinsic operation** is either an intrinsic unary operation or an intrinsic binary operation. An
 6 **intrinsic unary operation** is an operation of the form *intrinsic-operator* x_2 where x_2 is of an intrinsic
 7 type (4.4) listed in Table 7.3 for the unary intrinsic operator.

8 An **intrinsic binary operation** is an operation of the form x_1 *intrinsic-operator* x_2 where x_1 and
 9 x_2 are of the intrinsic types (4.4) listed in Table 7.3 for the binary intrinsic operator and are in shape
 10 conformance (7.1.10).

11 A **numeric intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is a numeric
 12 operator (+, −, *, /, or **). A **numeric intrinsic operator** is the operator in a numeric intrinsic
 13 operation.

14 The **character intrinsic operation** is the intrinsic operation for which the *intrinsic-operator* is (/) and
 15 both operands are of type character. The operands shall have the same kind type parameter. The
 16 **character intrinsic operator** is the operator in a character intrinsic operation.

17 A **logical intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is .AND., .OR.,
 18 .XOR., .NOT., .EQV., or .NEQV. and both operands are of type logical. A **logical intrinsic operator**
 19 is the operator in a logical intrinsic operation.

20 A **bits intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is //, .AND., .OR.,
 21 .XOR., .NOT., .EQV., or .NEQV. and at least one operand is of type bits. A **bits intrinsic operator**
 22 is the operator in a bits intrinsic operation.

23 A **relational intrinsic operator** is an *intrinsic-operator* that is .EQ., .NE., .GT., .GE., .LT., .LE.,
 24 ==, /=, >, >=, <, or <=. A **relational intrinsic operation** is an intrinsic operation for which the
 25 *intrinsic-operator* is a relational intrinsic operator. A **numeric relational intrinsic operation** is a
 26 relational intrinsic operation for which both operands are of numeric type. A **character relational**
 27 **intrinsic operation** is a relational intrinsic operation for which both operands are of type character.
 28 The kind type parameters of the operands of a character relational intrinsic operation shall be the same.
 29 A **bits relational intrinsic operation** is a relational intrinsic operation for which at least one of the
 30 operands is of type bits.

31 The interpretations defined in subclause 7.1.5 apply to both scalars and arrays; the interpretation for
 32 arrays is obtained by applying the interpretation for scalars element by element.

NOTE 7.15

For example, if X is of type real, J is of type integer, and INT is the real-to-integer intrinsic conversion function, the expression $\text{INT}(X + J)$ is an integer expression and $X + J$ is a real expression.

Table 7.3: **Type of operands and results for intrinsic operators**

Intrinsic operator <i>op</i>	Type of x_1	Type of x_2	Type of $[x_1] \text{ op } x_2$
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
	B	B	B
.EQ., .NE., ==, /=	I	I, R, Z, B	L, L, L, L
	R	I, R, Z, B	L, L, L, L
	Z	I, R, Z, B	L, L, L, L
	B	I, R, Z, B	L, L, L, L
	C	C	L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R, B	L, L, L
	R	I, R, B	L, L, L
	B	I, R, B	L, L, L
	C	C	L
.NOT.		L, B	L, B
.AND., .OR., .EQV., .NEQV., .XOR.	L	L	L
	B	B, I	B
	I	B	B
Note: The symbols I, R, Z, C, L, and B stand for the types integer, real, complex, character, logical, and bits, respectively. Where more than one type for x_2 is given, the type of the result of the operation is given in the same relative position in the next column.			

J3 internal note

Unresolved Technical Issue 104

Real-bits mixed-mode comparisons considered harmful.

The feature of intrinsic mixed-mode comparisons between BITS and REAL is, in my opinion, a serious misfeature. I am speaking particularly about .LT., .LE., .GT., and .GE..

I understand perfectly well that BITS work like unsigned integers for comparisons between themselves.

I understand perfectly well that it is very convenient to do mixed-mode comparisons with BITS and INTEGER, and that when doing so, converting to “unsigned” is a reasonable thing to do... though see next paragraph.

There are two schools of thought on this (what to convert mixed-mode operations between unsigned and signed integer to), “value-preserving” and “unsigned-preserving”. Interestingly enough the BITS proposal picked “unsigned-preserving” which is what most (not all) K&R C compilers did, and some K&R C compilers and the C standard picked “value-preserving”. This decision *might* profitably bear further consideration, but in any case is not the topic of this UTI!

What BITS has also added is mixed-mode REAL and BITS comparison. Here, bits don’t act like unsigned integers at all! In fact it is fair to say this operation doesn’t have any coherent mathematical theory behind it, being totally processor-dependent on the internal representation of said REALs.

Having “IF (X<1)” doing a totally different operation from “IF (X<1.)” is IMO an extraordinarily poor design, and likely to be error-prone to use in practice. It is also unnecessary; the three people in the world who actually want to sort a mixture of REALs and BITSs can do “IF (X<BITS(1.))” instead.

I *strongly* recommend removing this unnecessary feature, at least in the case of .LT. et al.

The case of .EQ./NE. comparison is not as striking as that of the other relational operators, but is also (IMO) an unnecessary frivolity that provides no functionality and some obscurity. Again, writing BITS() around the REAL or COMPLEX operand provides the functionality together with a visual indication that one is not doing a numerical comparison.

Hmm, actually with COMPLEX you have the further wrinkle that equality comparisons between complex and non-complex numerics have always promoted to complex with a zero imaginary part. Something which is again very different from what we have here.

Therefore I recommend that mixed-mode comparisons on numeric types (INTEGER, REAL, COMPLEX) only be extended for other numerical comparisons; when BITS are involved, that is only: BITS *op* BITS, BITS *op* INTEGER, and INTEGER *op* BITS.

I will further claim that since what is being done is not, with the exception of INTEGER, a numeric comparison, limiting it to the numeric types is inconsistent. But rather than extending .EQ./LT. to handle every data type (including derived types) we should limit them to what makes mathematical sense, i.e. what we had in F2003 + the bits/integer operations.

1 7.1.5.2 Numeric intrinsic operations

2 7.1.5.2.1 Interpretation of numeric intrinsic operations

3 The two operands of numeric intrinsic binary operations may be of different numeric types or different
 4 kind type parameters. Except for a value raised to an integer power, if the operands have different types
 5 or kind type parameters, the effect is as if each operand that differs in type or kind type parameter from
 6 those of the result is converted to the type and kind type parameter of the result before the operation
 7 is performed. When a value of type real or complex is raised to an integer power, the integer operand
 8 need not be converted.

9 A numeric operation is used to express a numeric computation. Evaluation of a numeric operation
 10 produces a numeric value. The permitted data types for operands of the numeric intrinsic operations
 11 are specified in 7.1.5.1.

- 1 The numeric operators and their interpretation in an expression are given in Table 7.4, where x_1 denotes
 2 the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.4: **Interpretation of the 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

- 3 The interpretation of a division operation depends on the types of the operands (7.1.5.2.2).
 4 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
 5 as the interpretation of $1/(x_1 ** \text{ABS}(x_2))$, which is subject to the rules of integer division (7.1.5.2.2).

NOTE 7.16

For example, $2 ** (-3)$ has the value of $1/(2 ** 3)$, which is zero.

6 7.1.5.2.2 Integer division

- 7 One operand of type integer may be divided by another operand of type integer. Although the math-
 8 ematical quotient of two integers is not necessarily an integer, Table 7.3 specifies that an expression
 9 involving the division operator with two operands of type integer is interpreted as an expression of type
 10 integer. The result of such an operation is the integer closest to the mathematical quotient and between
 11 zero and the mathematical quotient inclusively.

NOTE 7.17

For example, the expression $(-8) / 3$ has the value (-2) .

12 7.1.5.2.3 Complex exponentiation

- 13 In the case of a complex value raised to a complex power, the value of the operation $x_1 ** x_2$ is the
 14 principal value of $x_1^{x_2}$.

15 7.1.5.2.4 Evaluation of numeric intrinsic operations

- 16 Once the interpretation of a numeric intrinsic operation is established, the processor may evaluate any
 17 mathematically equivalent expression, provided that the integrity of parentheses is not violated.

- 18 Two expressions of a numeric type are mathematically equivalent if, for all possible values of their
 19 primaries, their mathematical values are equal. However, mathematically equivalent expressions of
 20 numeric type may produce different computational results.

NOTE 7.18

Any difference between the values of the expressions $(1./3.)*3.$ and $1.$ is a computational difference, not a mathematical difference. The difference between the values of the expressions $5/2$ and $5./2.$ is a mathematical difference, not a computational difference.

The mathematical definition of integer division is given in 7.1.5.2.2.

NOTE 7.19

The following are examples of expressions with allowable alternative forms that may 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.

<u>Expression</u>	<u>Allowable alternative form</u>
$X + Y$	$Y + X$
$X * Y$	$Y * X$
$-X + Y$	$Y - X$
$X + Y + Z$	$X + (Y + Z)$
$X - Y + Z$	$X - (Y - Z)$
$X * A / Z$	$X * (A / Z)$
$X * Y - X * Z$	$X * (Y - Z)$
$A / B / C$	$A / (B * C)$
$A / 5.0$	$0.2 * A$

The following are examples of expressions with forbidden alternative forms that shall not be used by a processor in the evaluation of those expressions.

<u>Expression</u>	<u>Forbidden alternative form</u>
$I / 2$	$0.5 * I$
$X * I / J$	$X * (I / J)$
$I / J / A$	$I / (J * A)$
$(X + Y) + Z$	$X + (Y + Z)$
$(X * Y) - (X * Z)$	$X * (Y - Z)$
$X * (Y - Z)$	$X * Y - X * Z$

- 1 The execution of any numeric operation whose result is not defined by the arithmetic used by the
- 2 processor is prohibited. Raising a negative-valued primary of type real to a real power is prohibited.

NOTE 7.20

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may 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 $(B - C)$ shall be evaluated and then added to A.

The inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions

$$A * I / J$$

$$A * (I / J)$$

may have different mathematical values if I and J are of type integer.

NOTE 7.21

Each operand in a numeric intrinsic operation has a type that may 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 may 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 **7.1.5.3 Character intrinsic operation**

2 **7.1.5.3.1 Interpretation of the character intrinsic operation**

3 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 The interpretation of the character intrinsic operator // when used to form an expression is given in
6 Table 7.6, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the
7 right of the operator.

Table 7.6: **Interpretation of the character intrinsic operator //**

Operator	Representing	Use of operator	Interpretation
//	Concatenation	$x_1 // x_2$	Concatenate x_1 with x_2

8 The result of the character intrinsic operation // 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 7.22

For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

11 **7.1.5.3.2 Evaluation of the character intrinsic operation**

12 A processor is only required to evaluate as much of the character intrinsic operation as is required by
13 the context in which the expression appears.

NOTE 7.23

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.

14 **7.1.5.4 Logical intrinsic operations**

1 7.1.5.4.1 Interpretation of logical intrinsic operations

2 A logical operation is used to express a logical computation. Evaluation of a logical operation produces
 3 a result of type logical. The permitted types for operands of the logical intrinsic operations are specified
 4 in 7.1.5.1.

5 The logical operators and their interpretation when used to form an expression are given in Table 7.7,
 6 where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the
 7 operator.

Table 7.7: 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
.XOR.	Logical nonequivalence	x_1 .XOR. x_2	True if either x_1 or x_2 is true, but not both

8 The values of the logical intrinsic operations are shown in Table 7.8.

Table 7.8: 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	x_1 .XOR. x_2
true	true	false	true	true	true	false	false
true	false	true	false	true	false	true	true
false	true	false	false	true	false	true	true
false	false	true	false	false	true	false	false

9 7.1.5.4.2 Evaluation of logical intrinsic operations

10 Once the interpretation of a logical intrinsic operation is established, the processor may evaluate any
 11 other expression that is logically equivalent, provided that the integrity of parentheses in any expression
 12 is not violated.

NOTE 7.24

For example, for the variables L1, L2, and L3 of type logical, the processor may choose to evaluate the expression

L1 .AND. L2 .AND. L3

as

L1 .AND. (L2 .AND. L3)

13 Two expressions of type logical are logically equivalent if their values are equal for all possible values of
 14 their primaries.

15 7.1.5.5 Bits intrinsic operations

1 7.1.5.5.1 Interpretation of bits intrinsic operations

2 For bits intrinsic operations other than concatenation (`//`), the two operands may be of different types
 3 or different kind type parameters. The effect is as if each operand that differs in type or kind type
 4 parameter from those of the result is converted to the type and kind type parameter of the result before
 5 the operation is performed.

6 Bit operations are used to express bitwise operations on sequences of bits, or to concatenate such
 7 sequences. Evaluation of a bits operation produces a result of type bits. The permitted types of
 8 operands of the bits intrinsic operations are specified in 7.1.5.1.

9 The bits operators and their interpretation when used to form an expression are given in Table 7.9,
 10 where x_1 denotes the operand of type bits to the left of the operator and x_2 denotes the operand of type
 11 bits to the right of the operator.

Table 7.9: Interpretation of the bits intrinsic operators

Operator	Representing	Use of operator	Interpretation
<code>//</code>	Concatenation	$x_1 // x_2$	Concatenation of x_1 and x_2
<code>.NOT.</code>	Bitwise NOT	<code>.NOT. x_2</code>	Bitwise NOT of x_2
<code>.AND.</code>	Bitwise AND	x_1 <code>.AND. x_2</code>	Bitwise AND of x_1 and x_2
<code>.OR.</code>	Bitwise inclusive OR	x_1 <code>.OR. x_2</code>	Bitwise OR of x_1 and x_2
<code>.EQV.</code>	Bitwise equivalence	x_1 <code>.EQV. x_2</code>	Bitwise equivalence of x_1 and x_2
<code>.NEQV.</code>	Bitwise nonequivalence	x_1 <code>.NEQV. x_2</code>	Bitwise nonequivalence of x_1 and x_2
<code>.XOR.</code>	Bitwise exclusive OR	x_1 <code>.XOR. x_2</code>	Bitwise exclusive OR of x_1 and x_2

12 The leftmost $\text{KIND}(x_1)$ bits of the result of the bits concatenation operation are the value of x_1 and the
 13 rightmost $\text{KIND}(x_2)$ bits of the result are the value of x_2 .

14 For a bits intrinsic operation other than `//`, the result value is computed separately for each pair of bits
 15 at corresponding positions in each operand. The values of the bits intrinsic operations, for bits denoted
 16 b_1 and b_2 , are shown in Table 7.10.

Table 7.10: The values of bits intrinsic operations other than `//`

b_1	b_2	<code>.NOT. b_2</code>	b_1 <code>.AND. b_2</code>	b_1 <code>.OR. b_2</code>	b_1 <code>.EQV. b_2</code>	b_1 <code>.NEQV. b_2</code>	b_1 <code>.XOR. b_2</code>
1	1	0	1	1	1	0	0
1	0	1	0	1	0	1	1
0	1	0	0	1	0	1	1
0	0	1	0	0	1	0	0

17 7.1.5.5.2 Evaluation of bits intrinsic operations

18 Once the interpretation of a bits operation is established, the processor may evaluate any other expression
 19 that is computationally equivalent, provided that the integrity of parentheses in any expression is not
 20 violated.

NOTE 7.25

For example, for the variables B1, B2, and B3 of type bits, the processor may choose to evaluate the expression

B1 `.XOR. B2` `.XOR. B3`

as

NOTE 7.25 (cont.)

B1 .XOR. (B2 .XOR. B3)

- 1 Two expressions of type bits are computationally equivalent if their values are equal for all possible
2 values of their primaries.

3 **7.1.5.6 Relational intrinsic operations**

4 **7.1.5.6.1 Interpretation of relational intrinsic operations**

- 5 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
7 operands of the relational intrinsic operators are specified in 7.1.5.1.

- 8 The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT.,
9 .LE., .GT., .GE., .EQ., and .NE., respectively.

- 10 If both operands of a bits relational operation do not have the same kind type parameter, the operand
11 with the smaller kind type parameter is converted to the same kind as the other operand. If one operand
12 of a bits relational operation is not of type bits, it is converted to type bits with the same kind type
13 parameter as the other operand. Any conversion takes place before the operation is evaluated.

NOTE 7.26

As shown in Table 7.3, 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 may 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.

- 14 Evaluation of a relational intrinsic operation produces a result of type default logical.
- 15 The interpretation of the relational intrinsic operators is given in Table 7.11, where x_1 denotes the
16 operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.11: **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

- 17 A numeric relational intrinsic operation is interpreted as having the logical value true if and only if the
18 values of the operands satisfy the relation specified by the operator.

In the numeric relational operation

1

2 $x_1 \text{ rel-op } x_2$

3 if the types or kind type parameters of x_1 and x_2 differ, their values are converted to the type and kind
4 type parameter of the expression $x_1 + x_2$ before evaluation.

5 A character relational intrinsic operation is interpreted as having the logical value true if and only if the
6 values of the operands satisfy the relation specified by the operator.

7 For a character relational intrinsic operation, the operands are compared one character at a time in
8 order, beginning with the first character of each character operand. If the operands are of unequal
9 length, the shorter operand is treated as if it were extended on the right with blanks to the length of
10 the longer operand. If both x_1 and x_2 are of zero length, x_1 is equal to x_2 ; if every character of x_1 is
11 the same as the character in the corresponding position in x_2 , x_1 is equal to x_2 . Otherwise, at the first
12 position where the character operands differ, the character operand x_1 is considered to be less than x_2
13 if the character value of x_1 at this position precedes the value of x_2 in the collating sequence (4.4.5.4);
14 x_1 is greater than x_2 if the character value of x_1 at this position follows the value of x_2 in the collating
15 sequence.

NOTE 7.27

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 types, the blank padding character is processor dependent.

16 A bits relational intrinsic operation is interpreted as having the logical value true if and only if the values
17 of the operands satisfy the relation specified by the operator.

18 For a bits relational intrinsic operation, x_1 and x_2 are equal if and only if each corresponding bit has
19 the same value. If x_1 and x_2 are not equal, and the leftmost unequal corresponding bit of x_1 is 1 and
20 x_2 is 0 then x_1 is greater than x_2 ; otherwise x_1 is less than x_2 .

21 **7.1.5.6.2 Evaluation of relational intrinsic operations**

22 Once the interpretation of a relational intrinsic operation is established, the processor may evaluate
23 any other expression that is relationally equivalent, provided that the integrity of parentheses in any
24 expression is not violated.

NOTE 7.28

For example, the processor may choose to evaluate the expression

$$I > J$$

where I and J are integer variables, as

$$J - I < 0$$

25 Two relational intrinsic operations are relationally equivalent if their logical values are equal for all
26 possible values of their primaries.

27 **7.1.6 Defined operations**

1 7.1.6.1 Definitions

2 A **defined operation** is either a defined unary operation or a defined binary operation. A **defined**
 3 **unary operation** is an operation that has the form *defined-unary-op* x_2 or *intrinsic-operator* x_2 and
 4 that is defined by a function and a generic interface (4.5.2, 12.4.3.4).

5 A function defines the unary operation op x_2 if

- 6 (1) the function is specified with a FUNCTION (12.6.2.2) or ENTRY (12.6.2.6) statement that
 7 specifies one dummy argument d_2 ,
- 8 (2) either
 - 9 (a) a generic interface (12.4.3.2) provides the function with a *generic-spec* of OPERA-
 10 TOR (op), or
 - 11 (b) there is a generic binding (4.5.2) in the declared type of x_2 with a *generic-spec* of
 12 OPERATOR (op) and there is a corresponding binding to the function in the dynamic
 13 type of x_2 ,
- 14 (3) the type of d_2 is compatible with the dynamic type of x_2 ,
- 15 (4) the type parameters, if any, of d_2 match the corresponding type parameters of x_2 , and
- 16 (5) either
 - 17 (a) the rank of x_2 matches that of d_2 or
 - 18 (b) the function is elemental and there is no other function that defines the operation.

19 If d_2 is an array, the shape of x_2 shall match the shape of d_2 .

20 A **defined binary operation** is an operation that has the form x_1 *defined-binary-op* x_2 or x_1 *intrinsic-*
 21 *operator* x_2 and that is defined by a function and a generic interface.

22 A function defines the binary operation x_1 op x_2 if

- 23 (1) the function is specified with a FUNCTION (12.6.2.2) or ENTRY (12.6.2.6) statement that
 24 specifies two dummy arguments, d_1 and d_2 ,
- 25 (2) either
 - 26 (a) a generic interface (12.4.3.2) provides the function with a *generic-spec* of OPERA-
 27 TOR (op), or
 - 28 (b) there is a generic binding (4.5.2) in the declared type of x_1 or x_2 with a *generic-*
 29 *spec* of OPERATOR (op) and there is a corresponding binding to the function in the
 30 dynamic type of x_1 or x_2 , respectively,
- 31 (3) the types of d_1 and d_2 are compatible with the dynamic types of x_1 and x_2 , respectively,
- 32 (4) the type parameters, if any, of d_1 and d_2 match the corresponding type parameters of x_1
 33 and x_2 , respectively, and
- 34 (5) either
 - 35 (a) the ranks of x_1 and x_2 match those of d_1 and d_2 or
 - 36 (b) the function is elemental, x_1 and x_2 are conformable, and there is no other function
 37 that defines the operation.

38 If d_1 or d_2 is an array, the shapes of x_1 and x_2 shall match the shapes of d_1 and d_2 , respectively.

NOTE 7.29

An intrinsic operator may be used as the operator in a defined operation. In such a case, the generic properties of the operator are extended.

1 An **extension operation** is a defined operation in which the operator is of the form *defined-unary-op*
 2 or *defined-binary-op*. Such an operator is called an **extension operator**. The operator used in an
 3 extension operation may be such that a generic interface for the operator may specify more than one
 4 function.

5 A **defined elemental operation** is a defined operation for which the function is elemental (12.8).

6 **7.1.6.2 Interpretation of a defined operation**

7 The interpretation of a defined operation is provided by the function that defines the operation.

8 The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT.,
 9 .LE., .GT., .GE., .EQ., and .NE., respectively.

10 **7.1.6.3 Evaluation of a defined operation**

11 Once the interpretation of a defined operation is established, the processor may evaluate any other
 12 expression that is equivalent, provided that the integrity of parentheses is not violated.

13 Two expressions of derived type are equivalent if their values are equal for all possible values of their
 14 primaries.

15 **7.1.7 Evaluation of operands**

16 It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely
 17 each operand, if the value of the expression can be determined otherwise.

NOTE 7.30

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.

18 If a statement contains a function reference in a part of an expression that need not be evaluated, all
 19 entities that would have become defined in the execution of that reference become undefined at the
 20 completion of evaluation of the expression containing the function reference.

NOTE 7.31

In the examples in Note 7.30, 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.

21 If a statement contains a function reference in a part of an expression that need not be evaluated, no
 22 invocation of that function in that part of the expression shall execute an image control statement other
 23 than CRITICAL or END CRITICAL.

NOTE 7.32

This restriction is intended to avoid inadvertant deadlock caused by optimization.

1 7.1.8 Integrity of parentheses

2 The rules for evaluation specified in subclause 7.1.5 state certain conditions under which a processor
3 may evaluate an expression that is different from the one specified by applying the rules given in 7.1.2
4 and rules for interpretation specified in subclause 7.1.5. However, any expression in parentheses shall be
5 treated as a data entity.

NOTE 7.33

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 $(A + B) - C$.

6 7.1.9 Type, type parameters, and shape of an expression

7 The type, type parameters, and shape of an expression depend on the operators and on the types, type
8 parameters, and shapes of the primaries used in the expression, and are determined recursively from
9 the syntactic form of the expression. The type of an expression is one of the intrinsic types (4.4) or a
10 derived type (4.5).

11 If an expression is a polymorphic primary or defined operation, the type parameters and the declared and
12 dynamic types of the expression are the same as those of the primary or defined operation. Otherwise
13 the type parameters and dynamic type of the expression are the same as its declared type and type
14 parameters; they are referred to simply as the type and type parameters of the expression.

```

15  R724  logical-expr      is  expr

```

16 C705 (R724) *logical-expr* shall be of type `logical`.

```

17  R725  char-expr      is expr

```

18 C706 (R725) *char-expr* shall be of type character.

```

19  R726  default-char-expr      is  expr

```

20 C707 (R726) *default-char-expr* shall be of type default character.

21 R727 *int-expr* is *expr*

22 C708 (R727) *int-expr* shall be of type integer.

23 R728 *numeric-expr* is *expr*

24 C709 (R728) *numeric-expr* shall be of type integer, real, or complex.

25 7.1.9.1 Type, type parameters, and shape of a primary

The type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, array constructor, structure constructor, function reference, type parameter inquiry, type parameter name, or parenthesized expression. If a primary is a constant, its type, type parameters, and shape are those of the constant. If it is a structure constructor, it is scalar and its type and type parameters are as described in 4.5.10. If it is an array constructor, its type, type parameters, and shape are as described in 4.7. If it is a variable or function reference, its type, type parameters, and shape are those of the variable (5.2, 5.3) or the function reference (12.5.3), respectively. If the function reference

1 is generic (12.4.3.2, 13.5) then its type, type parameters, and shape are those of the specific function
 2 referenced, which is determined by the types, type parameters, and ranks of its actual arguments as
 3 specified in 12.5.5.2. If it is a type parameter inquiry or type parameter name, it is a scalar integer with
 4 the kind of the type parameter.

5 If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expres-
 6 sion.

7 The associated target object is referenced if a pointer appears as

- 8 (1) a primary in an intrinsic or defined operation,
- 9 (2) the *expr* of a parenthesized primary, or
- 10 (3) the only primary on the right-hand side of an intrinsic assignment statement.

11 The type, type parameters, and shape of the primary are those of the current target. If the pointer is
 12 not associated with a target, it may appear as a primary only as an actual argument in a reference to
 13 a procedure whose corresponding dummy argument is declared to be a pointer, or as the target in a
 14 pointer assignment statement.

15 A disassociated array pointer or an unallocated allocatable array has no shape but does have rank.
 16 The type, type parameters, and rank of the result of the NULL intrinsic function depend on context
 17 (13.7.131).

18 7.1.9.2 Type, type parameters, and shape of the result of an operation

19 The type of the result of an intrinsic operation $[x_1] \text{ op } x_2$ is specified by Table 7.3. The shape of the
 20 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
 21 otherwise.

22 The type, type parameters, and shape of the result of a defined operation $[x_1] \text{ op } x_2$ are specified by the
 23 function defining the operation (7.1.6).

24 An expression of an intrinsic type has a kind type parameter. An expression of type character also has
 25 a character length parameter.

26 The type parameters of the result of an intrinsic operation are as follows.

- 27 (1) For an expression $x_1 // x_2$ where *//* is the character intrinsic operator and x_1 and x_2 are
 28 of type character, the character length parameter is the sum of the lengths of the operands
 29 and the kind type parameter is the kind type parameter of x_1 , which shall be the same as
 30 the kind type parameter of x_2 .
- 31 (2) For an expression $\text{op } x_2$ where *op* is an intrinsic unary operator and x_2 is of type integer,
 32 real, complex, logical, or bits, the kind type parameter of the expression is that of the
 33 operand.
- 34 (3) For an expression $x_1 \text{ op } x_2$ where *op* is a numeric intrinsic binary operator with one operand
 35 of type integer and the other of type real or complex, the kind type parameter of the
 36 expression is that of the real or complex operand.
- 37 (4) For an expression $x_1 \text{ op } x_2$ where *op* is a numeric intrinsic binary operator with both
 38 operands of the same type and kind type parameters, or with one real and one complex
 39 with the same kind type parameters, the kind type parameter of the expression is identical
 40 to that of each operand. In the case where both operands are integer with different kind type
 41 parameters, the kind type parameter of the expression is that of the operand with the greater
 42 decimal exponent range if the decimal exponent ranges are different; if the decimal exponent
 43 ranges are the same, the kind type parameter of the expression is processor dependent, but
 44 it is the same as that of one of the operands. In the case where both operands are any
 45 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.
- (5) 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.
- (6) For an expression $x_1 // x_2$ where both operands are of type bits, the kind type parameter of the expression is the sum of the kind type parameters of the operands.
- (7) For an expression $x_1 \text{ op } x_2$ where *op* is a bits intrinsic binary operator other than *//*, the kind type parameter of the expression is the maximum of the kind type parameters of x_1 and x_2 .
- (8) For an expression $x_1 \text{ op } x_2$ where *op* is a relational intrinsic operator, the expression has the default logical kind type parameter.
- C710 The kind type parameter of the result of a bits concatenation operation expression shall be a bits kind type parameter value supported by the processor.

7.1.10 Conformability rules for elemental operations

An **elemental operation** is an intrinsic operation or a defined elemental operation. Two entities are in **shape conformance** if both are arrays of the same shape, or one or both are scalars.

For all elemental binary operations, the two operands shall be in shape conformance. In the case where one is a scalar and the other an array, the scalar is treated as if it were an array of the same shape as the array operand with every element, if any, of the array equal to the value of the scalar.

7.1.11 Specification expression

A **specification expression** is an expression with limitations that make it suitable for use in specifications such as length type parameters (C404) and array bounds (R513, R514). A *specification-expr* shall be an initialization expression unless it is in an interface body (12.4.3.2), the specification part of a subprogram or BLOCK construct, or the *declaration-type-spec* of a FUNCTION statement (12.6.2.2).

R729 *specification-expr* **is** *scalar-int-expr*

C711 (R729) The *scalar-int-expr* shall be a restricted expression.

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

- (1) a constant or subobject of a constant,
- (2) an object designator with a base object that is a dummy argument that has neither the OPTIONAL nor the INTENT (OUT) attribute,
- (3) an object designator with a base object that is in a common block,
- (4) an object designator with a base object that is made accessible by use association or host association,
- (5) an object designator with a base object that is a local variable of the procedure containing the BLOCK construct in which the restricted expression appears,
- (6) an object designator with a base object that is a local variable of an outer BLOCK construct containing the BLOCK construct in which the restricted expression appears,
- (7) an array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-control* is a restricted expression,
- (8) a structure constructor where each component is a restricted expression,

- 1 (9) a specification inquiry where each designator or function argument is
 - 2 (a) a restricted expression or
 - 3 (b) a variable whose properties inquired about are not
 - 4 (i) dependent on the upper bound of the last dimension of an assumed-size array,
 - 5 (ii) deferred, or
 - 6 (iii) defined by an expression that is not a restricted expression,
- 7 (10) a reference to any other standard intrinsic function where each argument is a restricted expression,
- 8 (11) a reference to a specification function where each argument is a restricted expression,
- 9 (12) a type parameter of the derived type being defined,
- 10 (13) an *ac-do-variable* within an array constructor where each *scalar-int-expr* of the correspond-
 11 ing *ac-implied-do-control* is a restricted expression, or
- 12 (14) a restricted expression enclosed in parentheses,
- 13

14 where each subscript, section subscript, substring starting point, substring ending point, and type pa-
 15 rameter value is a restricted expression, and where any final subroutine that is invoked is pure.

16 A **specification inquiry** is a reference to

- 17 (1) an array inquiry function (13.5.7),
- 18 (2) the bit inquiry function BIT_SIZE,
- 19 (3) the character inquiry function LEN,
- 20 (4) the kind inquiry function KIND,
- 21 (5) the bits kind inquiry function BITS_KIND,
- 22 (6) the character inquiry function NEW_LINE,
- 23 (7) a numeric inquiry function (13.5.6),
- 24 (8) a type parameter inquiry (6.1.4),
- 25 (9) an IEEE inquiry function (14.9.1),
- 26 (10) the function C_SIZEOF from the intrinsic module ISO_C_BINDING (15.2.3.6) , or
- 27 (11) the COMPILER_VERSION or COMPILER_OPTIONS inquiry functions from the intrinsic
 28 module ISO_FORTRAN_ENV (13.8.2.3, 13.8.2.4).

29 A function is a **specification function** if it is a pure function, is not a standard intrinsic function, is
 30 not an internal function, is not a statement function, and does not have a dummy procedure argument.

31 Evaluation of a specification expression shall not directly or indirectly cause a procedure defined by the
 32 subprogram in which it appears to be invoked.

NOTE 7.34

Specification functions are nonintrinsic functions that may 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.

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

- 1 If a specification expression includes a specification inquiry that depends on a type parameter or an
 2 array bound of an entity specified in the same *specification-part*, the type parameter or array bound
 3 shall be specified in a prior specification of the *specification-part*. The prior specification may be to the
 4 left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*. If a
 5 specification expression includes a reference to the value of an element of an array specified in the same
 6 *specification-part*, the array shall be completely specified in prior declarations.
- 7 If a specification expression in a module or submodule includes a reference to a generic entity, that
 8 generic entity shall have no specific procedures defined in the module or submodule subsequent to the
 9 specification expression.

NOTE 7.35

The following are examples of specification expressions:

```

LBOUND (B, 1) + 5 ! B is an assumed-shape dummy array
M + LEN (C)      ! M and C are dummy arguments
2 * PRECISION (A) ! A is a real variable made accessible
                  ! by a USE statement

```

7.1.12 Initialization expression

11 An **initialization expression** is an expression with limitations that make it suitable for use as a kind
 12 type parameter, initializer, or named constant. It is an expression in which each operation is intrinsic,
 13 and each primary is

- 14 (1) a constant or subobject of a constant,
- 15 (2) an array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-control* is an initialization expression,
- 16 (3) a structure constructor where each *component-spec* corresponding to
 - 18 (a) an allocatable component is a reference to the intrinsic function NULL,
 - 19 (b) a pointer component is a reference to the intrinsic function NULL or an initialization target, and
 - 20 (c) any other component is an initialization expression,
- 21 (4) a reference to an elemental standard intrinsic function, where each argument is an initialization expression,
- 22 (5) a reference to a transformational standard intrinsic function other than NULL, where each argument is an initialization expression,
- 23 (6) A reference to the transformational 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 an initialization expression,
- 24 (7) a reference to the transformational function IEEE_SELECTED_REAL_KIND from the intrinsic module IEEE_ARITHMETIC (14), where each argument is an initialization expression,
- 25 (8) a specification inquiry where each designator or function argument is
 - 26 (a) an initialization expression or
 - 27 (b) a variable whose properties inquired about are not
 - 28 (i) assumed,
 - 29 (ii) deferred, or
 - 30 (iii) defined by an expression that is not an initialization expression,
- 31 (9) a kind type parameter of the derived type being defined,

- 1 (10) a *data-i-do-variable* within a *data-implied-do*,
 2 (11) an *ac-do-variable* within an array constructor where each *scalar-int-expr* of the correspond-
 3 ing *ac-implied-do-control* is an initialization expression, or
 4 (12) an initialization expression enclosed in parentheses,

5 and where each subscript, section subscript, substring starting point, substring ending point, and type
 6 parameter value is an initialization expression.

7 R730 *initialization-expr* **is** *expr*

8 C712 (R730) *initialization-expr* shall be an initialization expression.

9 R731 *char-initialization-expr* **is** *char-expr*

10 C713 (R731) *char-initialization-expr* shall be an initialization expression.

11 R732 *int-initialization-expr* **is** *int-expr*

12 C714 (R732) *int-initialization-expr* shall be an initialization expression.

13 R733 *logical-initialization-expr* **is** *logical-expr*

14 C715 (R733) *logical-initialization-expr* shall be an initialization expression.

15 If an initialization expression includes a specification inquiry that depends on a type parameter or an
 16 array bound of an entity specified in the same *specification-part*, the type parameter or array bound
 17 shall be specified in a prior specification of the *specification-part*. The prior specification may be to the
 18 left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*.

19 If an initialization expression in a module or submodule includes a reference to a generic entity, that
 20 generic entity shall have no specific procedures defined in the module or submodule subsequent to the
 21 initialization expression.

NOTE 7.36

The following are examples of initialization expressions:

```

3
-3 + 4
'AB'
'AB' // 'CD'
('AB' // 'CD') // 'EF'
SIZE (A)
DIGITS (X) + 4
4.0 * atan(1.0)
ceiling(number_of_decimal_digits / log10(radix(0.0)))

```

where A is an explicit-shaped array with constant bounds and X is of type default real.

7.2 Assignment

7.2.1 Assignment statement

A variable may be defined or redefined by execution of an assignment statement.

7.2.1.1 General form

- 1 R734 *assignment-stmt* is *variable* = *expr*
- 2 C716 (R734) The *variable* shall not be a whole assumed-size array.

NOTE 7.37

Examples of an assignment statement are:

```
A = 3.5 + X * Y
I = INT (A)
```

- 3 An *assignment-stmt* shall meet the requirements of either a defined assignment statement or an intrinsic
4 assignment statement.

7.2.1.2 Intrinsic assignment statement

- 6 An **intrinsic assignment statement** is an assignment statement that is not a defined assignment
7 statement (7.2.1.4). In an intrinsic assignment statement,

- 8 (1) if the variable is polymorphic it shall be allocatable,
9 (2) if *variable* is a co-indexed object, it shall not be of a type that has an allocatable ultimate
10 component,
11 (3) if *expr* is an array then the variable shall also be an array,
12 (4) the shapes of the variable and *expr* shall conform unless the variable is an allocatable array
13 that has the same rank as *expr* and is neither a co-array nor a co-indexed object,
14 (5) if the variable is an allocatable co-array or co-indexed object, it shall not be polymorphic,
15 (6) if the variable is polymorphic it shall be type compatible with *expr* and have the same rank;
16 otherwise the declared types of the variable and *expr* shall conform as specified in Table
17 7.12,
18 (7) if the variable is of derived type each kind type parameter of the variable shall have the
19 same value as the corresponding type parameter of *expr*, and
20 (8) if the variable is of derived type each length type parameter of the variable shall have the
21 same value as the corresponding type parameter of *expr* unless the variable is allocatable,
22 is not a co-array or co-indexed object, and its corresponding type parameter is deferred.

Table 7.12: **Type conformance for the intrinsic assignment statement**

Type of the variable	Type of <i>expr</i>
integer	integer, real, complex, bits
real	integer, real, complex, bits
complex	integer, real, complex, bits
ISO 10646, ASCII, or default character	ISO 10646, ASCII, or default character
other character	character of the same kind type parameter as the variable
logical	logical, bits
bits	integer, real, complex, bits
derived type	same derived type as the variable

- 24 A **numeric intrinsic assignment statement** is an intrinsic assignment statement for which the vari-
25 able and *expr* are of numeric type. A **character intrinsic assignment statement** is an intrinsic
26 assignment statement for which the variable and *expr* are of type character. A **logical intrinsic as-**
27 **signment statement** is an intrinsic assignment statement for which the variable and *expr* are of type
28 logical. A **bits intrinsic assignment statement** is an intrinsic assignment statement for which either
29 the variable or *expr* is of type bits. A **derived-type intrinsic assignment statement** is an intrinsic
assignment statement for which the variable and *expr* are of derived type.

1

2 An **array intrinsic assignment statement** is an intrinsic assignment statement for which the variable
3 is an array. The variable shall not be a many-one array section (6.2.2.2.2).

4 If the variable is a pointer, it shall be associated with a definable target such that the type, type
5 parameters, and shape of the target and *expr* conform.

6 7.2.1.3 Interpretation of intrinsic assignments

7 Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all
8 expressions within *variable* (7.1), the possible conversion of *expr* to the type and type parameters of the
9 variable (Table 7.13), and the definition of the variable with the resulting value. The execution of the
10 assignment shall have the same effect as if the evaluation of *expr* and the evaluation of all expressions
11 in *variable* occurred before any portion of the variable is defined by the assignment. The evaluation of
12 expressions within *variable* shall neither affect nor be affected by the evaluation of *expr*. No value is
13 assigned to the variable if it is of type character and zero length, or is an array of size zero.

14 If the variable is a pointer, the value of *expr* is assigned to the target of the variable.

15 If the variable is an allocated allocatable variable, it is deallocated if *expr* is an array of different shape,
16 any of the corresponding length type parameter values of the variable and *expr* differ, or the variable
17 is polymorphic and the dynamic type of the variable and *expr* differ. If the variable is or becomes an
18 unallocated allocatable variable, then it is allocated with each deferred type parameter equal to the
19 corresponding type parameter of *expr*, with the shape of *expr*, with each lower bound equal to the
20 corresponding element of LBOUND(*expr*), and if the variable is polymorphic, with the same dynamic
21 type as *expr*.

NOTE 7.38

For example, given the declaration

```
CHARACTER(:),ALLOCATABLE :: NAME
```

then after the assignment statement

```
NAME = 'Dr. '//FIRST_NAME//' '//SURNAME
```

NAME will have the length LEN(FIRST_NAME)+LEN(SURNAME)+5, even if it had previously been unallocated, or allocated with a different length. However, for the assignment statement

```
NAME(:) = 'Dr. '//FIRST_NAME//' '//SURNAME
```

NAME must already be allocated at the time of the assignment; the assigned value is truncated or blank padded to the previously allocated length of NAME.

22 Both *variable* and *expr* may contain references to any portion of the variable.

NOTE 7.39

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

NOTE 7.39 (cont.)

value following the assignment is 'AABCDF'.

- 1 If *expr* is a scalar and the variable is an array, the *expr* is treated as if it were an array of the same
- 2 shape as the variable with every element of the array equal to the scalar value of *expr*.
- 3 If the variable is an array, the assignment is performed element-by-element on corresponding array
- 4 elements of the variable and *expr*.

NOTE 7.40

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 The processor may perform the element-by-element assignment in any order.

NOTE 7.41

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 For a numeric intrinsic assignment statement, the variable and *expr* may have different numeric types
- 7 or different kind type parameters, 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 7.13.

Table 7.13: **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: The functions INT, REAL, CMPLX, and KIND are the generic functions defined in 13.7.	

- 9 For a logical intrinsic assignment statement, the variable and *expr* may have different kind type param-
- 10 eters, in which case the value of *expr* is converted to the kind type parameter of the variable.
- 11 For a character intrinsic assignment statement, the variable and *expr* may have different character length
- 12 parameters in which case the conversion of *expr* to the length of the variable is as follows.

- 1 (1) If the length of the variable is less than that of *expr*, the value of *expr* is truncated from
 2 the right until it is the same length as the variable.
- 3 (2) If the length of the variable is greater than that of *expr*, the value of *expr* is extended on
 4 the right with blanks until it is the same length as the variable.
- 5 If the variable and *expr* have different kind type parameters, each character *c* in *expr* is converted to
 6 the kind type parameter of the variable by ACHAR(IACHAR(*c*),KIND(*variable*)).

NOTE 7.42

For nondefault character types, 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.

- 7 For a bits intrinsic assignment statement, the variable and *expr* may have different types or different
 8 kind type parameters, in which case the value of *expr* is converted to the type and kind type parameter
 9 of the variable according to the rules of Table 7.14.

Table 7.14: **Bits 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>))
logical	LOGICAL (<i>expr</i> , KIND = KIND (<i>variable</i>))
bits	BITS (<i>expr</i> , KIND = KIND (<i>variable</i>))
Note: The functions BITS, INT, REAL, CMPLX, LOGICAL, and KIND are the generic functions defined in 13.7.	

NOTE 7.43

Bits assignment is not always the same as the result of the TRANSFER intrinsic, because:

- bits assignment operates elementally, whereas TRANSFER does not preserve array element boundaries;
- for scalars, if the source is larger TRANSFER uses those bits which occur first in memory whereas bits assignment always uses the “rightmost” bits (according to the model for bits values), independent of the endianness of the processor’s memory addressing;
- if the source is smaller, TRANSFER uses it for the part of the result which occurs first in memory address order and leaves the rest of the result processor-dependent, whereas bits assignment copies the source to the rightmost bits and makes the remaining bits all zero.

- 10 A derived-type intrinsic assignment is performed as if each component of the variable were assigned
 11 from the corresponding component of *expr* using pointer assignment (7.2.2) for each pointer component,
 12 defined assignment for each nonpointer nonallocatable component of a type that has a type-bound defined
 13 assignment consistent with the component, intrinsic assignment for each other nonpointer nonallocatable
 14 component, and intrinsic assignment for each allocated co-array component. For unallocated co-array
 15 components, the corresponding component of the variable shall be unallocated. For a non-co-array
 16 allocatable component the following sequence of operations is applied.

- 17 (1) If the component of the variable is allocated, it is deallocated.
- 18 (2) If the component of the value of *expr* is allocated, the corresponding component of the
 19 variable is allocated with the same dynamic type and type parameters as the component

1 of the value of *expr*. If it is an array, it is allocated with the same bounds. The value of
 2 the component of the value of *expr* is then assigned to the corresponding component of the
 3 variable using defined assignment if the declared type of the component has a type-bound
 4 defined assignment consistent with the component, and intrinsic assignment for the dynamic
 5 type of that component otherwise.

6 The processor may perform the component-by-component assignment in any order or by any means that
 7 has the same effect.

NOTE 7.44

For an example of a derived-type intrinsic assignment statement, if C and D are of the same derived type with a pointer component P and nonpointer components S, T, U, and V of type integer, logical, character, and another derived type, respectively, the intrinsic

$$C = D$$

pointer assigns D%P to C%P. It assigns D%S to C%S, D%T to C%T, and D%U to C%U using intrinsic assignment. It assigns D%V to C%V using defined assignment if objects of that type have a compatible type-bound defined assignment, and intrinsic assignment otherwise.

NOTE 7.45

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.

8 7.2.1.4 Defined assignment statement

9 A **defined assignment statement** is an assignment statement that is defined by a subroutine and a
 10 generic interface (4.5.2, 12.4.3.4.3) that specifies ASSIGNMENT (=). A **defined elemental assign-**
 11 **ment statement** is a defined assignment statement for which the subroutine is elemental (12.8).

12 A subroutine defines the defined assignment $x_1 = x_2$ if

- 13 (1) the subroutine is specified with a SUBROUTINE (12.6.2.3) or ENTRY (12.6.2.6) statement
 14 that specifies two dummy arguments, d_1 and d_2 ,
- 15 (2) either
 - 16 (a) a generic interface (12.4.3.2) provides the subroutine with a *generic-spec* of ASSIGN-
 17 MENT (=), or
 - 18 (b) there is a generic binding (4.5.2) in the declared type of x_1 or x_2 with a *generic-spec*
 19 of ASSIGNMENT (=) and there is a corresponding binding to the subroutine in the
 20 dynamic type of x_1 or x_2 , respectively,
- 21 (3) the types of d_1 and d_2 are compatible with the dynamic types of x_1 and x_2 , respectively,
- 22 (4) the type parameters, if any, of d_1 and d_2 match the corresponding type parameters of x_1
 23 and x_2 , respectively, and
- 24 (5) either
 - 25 (a) the ranks of x_1 and x_2 match those of d_1 and d_2 or
 - 26 (b) the subroutine is elemental, x_1 and x_2 are conformable, and there is no other subrou-
 27 tine that defines the operation.

28 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.

29 7.2.1.5 Interpretation of defined assignment statements

1

2 R737 *bounds-spec* **is** *lower-bound-expr* :3 R738 *bounds-remapping* **is** *lower-bound-expr* : *upper-bound-expr*4 R739 *data-target* **is** *variable*5 **or** *expr*6 C726 (R739) A *variable* shall have either the TARGET or POINTER attribute, and shall not be an
7 array section with a vector subscript.8 C727 (R739) A *data-target* shall not be a co-indexed object.**NOTE 7.47**

A data pointer and its target are always on the same image. A co-array may be of a derived type with pointer or allocatable subcomponents. For example, if PTR is a pointer component, Z[P]%PTR is a reference to the target of component PTR of Z on image P. This target is on image P and its association with Z[P]%PTR must have been established by the execution of an ALLOCATE statement or a pointer assignment on image P.

9 C728 (R739) An *expr* shall be a reference to a function whose result is a data pointer.10 R740 *proc-pointer-object* **is** *proc-pointer-name*11 **or** *proc-component-ref*12 R741 *proc-component-ref* **is** *scalar-variable* % *procedure-component-name*13 C729 (R741) The *scalar-variable* shall be a *data-ref*.14 C730 (R741) The *procedure-component-name* shall be the name of a procedure pointer component of
15 the declared type of *scalar-variable*.16 R742 *proc-target* **is** *expr*17 **or** *procedure-name*18 **or** *proc-component-ref*19 C731 (R742) An *expr* shall be a reference to a function whose result is a procedure pointer.20 C732 (R742) A *procedure-name* shall be the name of an external, internal, module, or dummy proce-
21 dure, a procedure pointer, or a specific intrinsic function listed in 13.6 and not marked with a
22 bullet (●).23 C733 (R742) The *proc-target* shall not be a nonintrinsic elemental procedure.24 **7.2.2.3 Data pointer assignment**25 If *data-pointer-object* is not polymorphic (4.3.1.3) and *data-target* is polymorphic with dynamic type
26 that differs from its declared type, the assignment target is the ancestor component of *data-target* that
27 has the type of *data-pointer-object*. Otherwise, the assignment target is *data-target*.28 If *data-target* is not a pointer, *data-pointer-object* becomes pointer associated with the assignment target.
29 Otherwise, the pointer association status of *data-pointer-object* becomes that of *data-target*; if *data-target*
30 is associated with an object, *data-pointer-object* becomes associated with the assignment target. If *data-*
31 *target* is allocatable, it shall be allocated.32 If *data-pointer-object* is polymorphic, it assumes the dynamic type of *data-target*. If *data-pointer-object*
33 is of sequence derived type or a type with the BIND attribute, the dynamic type of *data-target* shall be

- 1 that derived type.
- 2 If *data-target* is a disassociated pointer, all nondeferred type parameters of the declared type of *data-*
 3 *pointer-object* that correspond to nondeferred type parameters of *data-target* shall have the same values
 4 as the corresponding type parameters of *data-target*.
- 5 Otherwise, all nondeferred type parameters of the declared type of *data-pointer-object* shall have the
 6 same values as the corresponding type parameters of *data-target*.
- 7 If *data-pointer-object* has nondeferred type parameters that correspond to deferred type parameters of
 8 *data-target*, *data-target* shall not be a pointer with undefined association status.
- 9 If *data-pointer-object* has the CONTIGUOUS attribute, *data-target* shall be contiguous.
- 10 If *bounds-remapping-list* is specified, *data-target* shall be contiguous (5.3.6) or of rank one. It shall not
 11 be a disassociated or undefined pointer, and the size of *data-target* shall not be less than the size of
 12 *data-pointer-object*. The elements of the target of *data-pointer-object*, in array element order (6.2.2.1),
 13 are the first SIZE(*data-pointer-object*) elements of *data-target*.
- 14 If no *bounds-remapping-list* is specified, the extent of a dimension of *data-pointer-object* is the extent of
 15 the corresponding dimension of *data-target*. If *bounds-spec-list* appears, it specifies the lower bounds;
 16 otherwise, the lower bound of each dimension is the result of the intrinsic function LBOUND (13.7.97)
 17 applied to the corresponding dimension of *data-target*. The upper bound of each dimension is one less
 18 than the sum of the lower bound and the extent.

19 7.2.2.4 Procedure pointer assignment

- 20 If the *proc-target* is not a pointer, *proc-pointer-object* becomes pointer associated with *proc-target*. Other-
 21 wise, the pointer association status of *proc-pointer-object* becomes that of *proc-target*; if *proc-target* is
 22 associated with a procedure, *proc-pointer-object* becomes associated with the same procedure.
- 23 If *proc-target* is the name of an internal procedure the **host instance** of *proc-pointer-object* becomes
 24 the innermost currently executing instance of the host procedure. Otherwise if *proc-target* has a host
 25 instance the host instance of *proc-pointer-object* becomes that instance. Otherwise *proc-pointer-object*
 26 has no host instance.
- 27 If *proc-pointer-object* has an explicit interface, its characteristics shall be the same as *proc-target* except
 28 that *proc-target* may be pure even if *proc-pointer-object* is not pure and *proc-target* may be an elemental
 29 intrinsic procedure even if *proc-pointer-object* is not elemental.
- 30 If the characteristics of *proc-pointer-object* or *proc-target* are such that an explicit interface is required,
 31 both *proc-pointer-object* and *proc-target* shall have an explicit interface.
- 32 If *proc-pointer-object* has an implicit interface and is explicitly typed or referenced as a function, *proc-*
 33 *target* shall be a function. If *proc-pointer-object* has an implicit interface and is referenced as a subroutine,
 34 *proc-target* shall be a subroutine.
- 35 If *proc-target* and *proc-pointer-object* are functions, they shall have the same type; corresponding type
 36 parameters shall either both be deferred or both have the same value.
- 37 If *procedure-name* is a specific procedure name that is also a generic name, only the specific procedure
 38 is associated with pointer-object.

39 7.2.2.5 Examples

NOTE 7.48

The following are examples of pointer assignment statements. (See Note 12.16 for declarations of P and BESSEL.)

```
NEW_NODE % LEFT => CURRENT_NODE
SIMPLE_NAME => TARGET_STRUCTURE % SUBSTRUCT % COMPONENT
PTR => NULL ( )
ROW => MAT2D (N, :)
WINDOW => MAT2D (I-1:I+1, J-1:J+1)
POINTER_OBJECT => POINTER_FUNCTION (ARG_1, ARG_2)
EVERY_OTHER => VECTOR (1:N:2)
WINDOW2 (0:, 0:) => MAT2D (ML:MU, NL:NU)
! P is a procedure pointer and BESSEL is a procedure with a
! compatible interface.
P => BESSEL

! Likewise for a structure component.
STRUCT % COMPONENT => BESSEL
```

NOTE 7.49

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
```

1 7.2.3 Masked array assignment – WHERE

2 7.2.3.1 General form of the masked array assignment

3 A **masked array assignment** is either a WHERE statement or a WHERE construct. It is used to
 4 mask the evaluation of expressions and assignment of values in array assignment statements, according
 5 to the value of a logical array expression.

6 R743 *where-stmt* **is** WHERE (*mask-expr*) *where-assignment-stmt*

7 R744 *where-construct* **is** *where-construct-stmt*
 8 [*where-body-construct*] ...


```

1      [ masked-elsewhere-stmt
2          [ where-body-construct ] ... ] ...
3      [ elsewhere-stmt
4          [ where-body-construct ] ... ]
5      end-where-stmt

```

6 R745 *where-construct-stmt* **is** *[where-construct-name:]* WHERE (*mask-expr*)

7	R746	<i>where-body-construct</i>	is	<i>where-assignment-stmt</i>
8			or	<i>where-stmt</i>
9			or	<i>where-construct</i>

10 R747 *where-assignment-stmt* is *assignment-stmt*

11 R748 *mask-expr* is *logical-expr*

12 R749 *masked-elsewhere-stmt* **is** ELSEWHERE (*mask-expr*) [*where-construct-name*]

13 R750 *elsewhere-stmt* is ELSEWHERE [*where-construct-name*]

14 R751 *end-where-stmt* **is** END WHERE [*where-construct-name*]

15 C734 (R747) A *where-assignment-stmt* that is a defined assignment shall be elemental.

C735 (R744) If the *where-construct-stmt* is identified by a *where-construct-name*, the corresponding
 17 *end-where-stmt* shall specify the same *where-construct-name*. If the *where-construct-stmt* is
 18 not identified by a *where-construct-name*, the corresponding *end-where-stmt* shall not specify
 19 a *where-construct-name*. If an *elsewhere-stmt* or a *masked-elsewhere-stmt* is identified by a
 20 *where-construct-name*, the corresponding *where-construct-stmt* shall specify the same *where-*
 21 *construct-name*.

22 C736 (R746) A statement that is part of a *where-body-construct* shall not be a branch target statement.

23 If a *where-construct* contains a *where-stmt*, a *masked-elsewhere-stmt*, or another *where-construct* then
 24 each *mask-expr* within the *where-construct* shall have the same shape. In each *where-assignment-stmt*,
 25 the *mask-expr* and the variable being defined shall be arrays of the same shape.

NOTE 7.50

Examples of a masked array assignment are:

```

WHERE (TEMP > 100.0) TEMP = TEMP - REDUCE_TEMP
WHERE (PRESSURE <= 1.0)
    PRESSURE = PRESSURE + INC_PRESSURE
    TEMP = TEMP - 5.0
ELSEWHERE
    RAINING = .TRUE.
END WHERE

```

26 7.2.3.2 Interpretation of masked array assignments

When a WHERE statement or a *where-construct-stmt* is executed, a control mask is established. In addition, when a WHERE construct statement is executed, a pending control mask is established. If the statement does not appear as part of a *where-body-construct*, the *mask-expr* of the statement is evaluated, and the control mask is established to be the value of *mask-expr*. The pending control mask is established to have the value .NOT. *mask-expr* upon execution of a WHERE construct statement that does not appear as part of a *where-body-construct*. The *mask-expr* is evaluated only once.

- 1 Each statement in a WHERE construct is executed in sequence.
- 2 Upon execution of a *masked-elsewhere-stmt*, the following actions take place in sequence.
 - 3 (1) The control mask m_c is established to have the value of the pending control mask.
 - 4 (2) The pending control mask is established to have the value m_c .AND. (.NOT. *mask-expr*).
 - 5 (3) The control mask m_c is established to have the value m_c .AND. *mask-expr*.
- 6 The *mask-expr* is evaluated at most once.
- 7 Upon execution of an ELSEWHERE statement, the control mask is established to have the value of the
- 8 pending control mask. No new pending control mask value is established.
- 9 Upon execution of an ENDWHERE statement, the control mask and pending control mask are es-
- 10 tablished to have the values they had prior to the execution of the corresponding WHERE construct
- 11 statement. Following the execution of a WHERE statement that appears as a *where-body-construct*, the
- 12 control mask is established to have the value it had prior to the execution of the WHERE statement.

NOTE 7.51

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.

- 13 Upon execution of a WHERE construct statement that is part of a *where-body-construct*, the pending
- 14 control mask is established to have the value m_c .AND. (.NOT. *mask-expr*). The control mask is then
- 15 established to have the value m_c .AND. *mask-expr*. The *mask-expr* is evaluated at most once.
- 16 Upon execution of a WHERE statement that is part of a *where-body-construct*, the control mask is
- 17 established to have the value m_c .AND. *mask-expr*. The pending mask is not altered.
- 18 If a nonelemental function reference occurs in the *expr* or *variable* of a *where-assignment-stmt* or in a
- 19 *mask-expr*, the function is evaluated without any masked control; that is, all of its argument expressions
- 20 are fully evaluated and the function is fully evaluated. If the result is an array and the reference is not
- 21 within the argument list of a nonelemental function, elements corresponding to true values in the control
- 22 mask are selected for use in evaluating the *expr*, *variable* or *mask-expr*.
- 23 If an elemental operation or function reference occurs in the *expr* or *variable* of a *where-assignment-stmt*
- 24 or in a *mask-expr*, and is not within the argument list of a nonelemental function reference, the operation
- 25 is performed or the function is evaluated only for the elements corresponding to true values of the control
- 26 mask.
- 27 If an array constructor appears in a *where-assignment-stmt* or in a *mask-expr*, the array constructor is

- 1 evaluated without any masked control and then the *where-assignment-stmt* is executed or the *mask-expr*
 2 is evaluated.
- 3 When a *where-assignment-stmt* is executed, the values of *expr* that correspond to true values of the
 4 control mask are assigned to the corresponding elements of the variable.
- 5 The value of the control mask is established by the execution of a WHERE statement, a WHERE con-
 6 struct statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE
 7 statement. Subsequent changes to the value of entities in a *mask-expr* have no effect on the value of the
 8 control mask. The execution of a function reference in the mask expression of a WHERE statement is
 9 permitted to affect entities in the assignment statement.

NOTE 7.52

Examples of function references in masked array assignments are:

```
WHERE (A > 0.0)
A = LOG (A)           ! LOG is invoked only for positive elements.
A = A / SUM (LOG (A)) ! LOG is invoked for all elements
                     ! because SUM is transformational.
END WHERE
```

7.2.4 FORALL**7.2.4.1 Form of the FORALL Construct**

The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements to be controlled by a single *forall-triplet-spec-list* and *scalar-mask-expr*.

R752	<i>forall-construct</i>	is	<i>forall-construct-stmt</i> [<i>forall-body-construct</i>] ... <i>end-forall-stmt</i>
R753	<i>forall-construct-stmt</i>	is	[<i>forall-construct-name</i> :] FORALL <i>forall-header</i>
R754	<i>forall-header</i>	is	([<i>type-spec</i> ::] <i>forall-triplet-spec-list</i> [, <i>scalar-mask-expr</i>])
R755	<i>forall-triplet-spec</i>	is	<i>index-name</i> = <i>subscript</i> : <i>subscript</i> [: <i>stride</i>]
R619	<i>subscript</i>	is	<i>scalar-int-expr</i>
R622	<i>stride</i>	is	<i>scalar-int-expr</i>
R756	<i>forall-body-construct</i>	is	<i>forall-assignment-stmt</i> or <i>where-stmt</i> or <i>where-construct</i> or <i>forall-construct</i> or <i>forall-stmt</i>
R757	<i>forall-assignment-stmt</i>	is	<i>assignment-stmt</i> or <i>pointer-assignment-stmt</i>
R758	<i>end-forall-stmt</i>	is	END FORALL [<i>forall-construct-name</i>]

(R758) If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have the same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-*

- 1 *construct-stmt* shall have the same *forall-construct-name*.
- 2 C738 (R754) *type-spec* shall specify type integer.
- 3 C739 (R754) The *scalar-mask-expr* shall be scalar and of type logical.
- 4 C740 (R754) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined
5 operation, shall be a pure procedure (12.7).
- 6 C741 (R755) The *index-name* shall be a named scalar variable of type integer.
- 7 C742 (R755) A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-*
8 *name* in the *forall-triplet-spec-list* in which it appears.
- 9 C743 (R756) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-*
10 *construct*.
- 11 C744 (R756) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined
12 operation, assignment, or finalization, shall be a pure procedure.
- 13 C745 (R756) A *forall-body-construct* shall not be a branch target.
- 14 C746 (R757) The *variable* in an *assignment-stmt* that is a *forall-assignment-stmt* shall be a *designator*.

NOTE 7.53

An example of a FORALL construct is:

```
REAL :: A(10, 10), B(10, 10) = 1.0
. . .
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0.0)
  A(I, J) = REAL (I + J - 2)
  B(I, J) = A(I, J) + B(I, J) * REAL (I * J)
END FORALL
```

NOTE 7.54

An assignment statement that is a FORALL body construct may be a scalar or array assignment statement, or a defined assignment statement. The variable being defined will normally use each index name in the *forall-triplet-spec-list*. For example

```
FORALL (I = 1:N, J = 1:N)
  A(:, I, :, J) = 1.0 / REAL(I + J - 1)
END FORALL
```

broadcasts scalar values to rank-two subarrays of A.

NOTE 7.55

An example of a FORALL construct containing a pointer assignment statement is:

```
TYPE ELEMENT
  REAL ELEMENT_WT
  CHARACTER (32), POINTER :: NAME
END TYPE ELEMENT
TYPE(ELEMENT) CHART(200)
REAL WEIGHTS (1000)
```

NOTE 7.55 (cont.)

```

CHARACTER (32), TARGET :: NAMES (1000)
. . .
FORALL (I = 1:200, WEIGHTS (I + N - 1) > .5)
    CHART(I) % ELEMENT_WT = WEIGHTS (I + N - 1)
    CHART(I) % NAME => NAMES (I + N - 1)
END FORALL

```

The results of this FORALL construct cannot be achieved with a WHERE construct because a pointer assignment statement is not permitted in a WHERE construct.

- 1 An *index-name* in a *forall-construct* has a scope of the construct (16.4). It is a scalar variable that has
- 2 the type and type parameters that it would have if it were the name of a variable in the scoping unit
- 3 that includes the FORALL, and this type shall be integer type; it has no other attributes.

NOTE 7.56

The use of *index-name* variables in a FORALL construct does not affect variables of the same name, for example:

```

INTEGER :: X = -1
REAL A(5, 4)
J = 100
. . .
FORALL (X = 1:5, J = 1:4)
    A (X, J) = J
END FORALL

```

After execution of the FORALL, the variables X and J have the values -1 and 100 and A has the value

```

1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

```

4 7.2.4.2 Execution of the FORALL construct

5 7.2.4.2.1 Execution stages

6 There are three stages in the execution of a FORALL construct:

- 7 (1) determination of the values for *index-name* variables,
- 8 (2) evaluation of the *scalar-mask-expr*, and
- 9 (3) execution of the FORALL body constructs.

10 7.2.4.2.2 Determination of the values for index variables

11 The subscript and stride expressions in the *forall-triplet-spec-list* are evaluated. These expressions may
 12 be evaluated in any order. The set of values that a particular *index-name* variable assumes is determined
 13 as follows.

- 14 (1) The lower bound m_1 , the upper bound m_2 , and the stride m_3 are of type integer with the
 15 same kind type parameter as the *index-name*. Their values are established by evaluating

1 the first subscript, the second subscript, and the stride expressions, respectively, including,
 2 if necessary, conversion to the kind type parameter of the *index-name* according to the rules
 3 for numeric conversion (Table 7.13). If a stride does not appear, m_3 has the value 1. The
 4 value m_3 shall not be zero.

5 (2) Let the value of *max* be $(m_2 - m_1 + m_3)/m_3$. If $max \leq 0$ for some *index-name*, the execution
 6 of the construct is complete. Otherwise, the set of values for the *index-name* is

7
$$m_1 + (k - 1) \times m_3 \quad \text{where } k = 1, 2, \dots, max.$$

8 The set of combinations of *index-name* values is the Cartesian product of the sets defined by each triplet
 9 specification. An *index-name* becomes defined when this set is evaluated.

10 7.2.4.2.3 Evaluation of the mask expression

11 The *scalar-mask-expr*, if any, is evaluated for each combination of *index-name* values. If there is no
 12 *scalar-mask-expr*, it is as if it appeared with the value true. The *index-name* variables may be primaries
 13 in the *scalar-mask-expr*.

14 The **active combination** of *index-name* values is defined to be the subset of all possible combinations
 15 (7.2.4.2.2) for which the *scalar-mask-expr* has the value true.

NOTE 7.57

The *index-name* variables may appear in the mask, for example

```
FORALL (I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)
. . .
```

16 7.2.4.2.4 Execution of the FORALL body constructs

17 The *forall-body-constructs* are executed in the order in which they appear. Each construct is executed
 18 for all active combinations of the *index-name* values with the following interpretation:

19 Execution of a *forall-assignment-stmt* that is an *assignment-stmt* causes the evaluation of *expr* and all
 20 expressions within *variable* for all active combinations of *index-name* values. These evaluations may be
 21 done in any order. After all these evaluations have been performed, each *expr* value is assigned to the
 22 corresponding *variable*. The assignments may occur in any order.

23 Execution of a *forall-assignment-stmt* that is a *pointer-assignment-stmt* causes the evaluation of all
 24 expressions within *data-target* and *data-pointer-object* or *proc-target* and *proc-pointer-object*, the de-
 25 termination of any pointers within *data-pointer-object* or *proc-pointer-object*, and the determination of
 26 the target for all active combinations of *index-name* values. These evaluations may be done in any
 27 order. After all these evaluations have been performed, each *data-pointer-object* or *proc-pointer-object*
 28 is associated with the corresponding target. These associations may occur in any order.

29 In a *forall-assignment-stmt*, a defined assignment subroutine shall not reference any *variable* that be-
 30 comes defined by the statement.

NOTE 7.58

The following FORALL construct contains two assignment statements. The assignment to array B uses the values of array A computed in the previous statement, not the values A had prior to execution of the FORALL.

```
FORALL (I = 2:N-1, J = 2:N-1 )
  A (I, J) = A(I, J-1) + A(I, J+1) + A(I-1, J) + A(I+1, J)
  B (I, J) = 1.0 / A(I, J)
```

NOTE 7.58 (cont.)

```
END FORALL
```

Computations that would otherwise cause error conditions can be avoided by using an appropriate *scalar-mask-expr* that limits the active combinations of the *index-name* values. For example:

```
FORALL (I = 1:N, Y(I) /= 0.0)
  X(I) = 1.0 / Y(I)
END FORALL
```

- 1 Each statement in a *where-construct* (7.2.3) within a *forall-construct* is executed in sequence. When
- 2 a *where-stmt*, *where-construct-stmt* or *masked-elsewhere-stmt* is executed, the statement's *mask-expr* is
- 3 evaluated for all active combinations of *index-name* values as determined by the outer *forall-constructs*,
- 4 masked by any control mask corresponding to outer *where-constructs*. Any *where-assignment-stmt* is
- 5 executed for all active combinations of *index-name* values, masked by the control mask in effect for the
- 6 *where-assignment-stmt*.

NOTE 7.59

This FORALL construct contains a WHERE statement and an assignment statement.

```
INTEGER A(5,4), B(5,4)
FORALL ( I = 1:5 )
  WHERE ( A(I,:) == 0 ) A(I,:) = I
  B (I,:) = I / A(I,:)
END FORALL
```

When executed with the input array

```

      0  0  0  0
      1  1  1  0
A  =  2  2  0  2
      1  0  2  3
      0  0  0  0
```

the results will be

```

      1  1  1  1
      1  1  1  2
A  =  2  2  3  2
      1  4  2  3
      5  5  5  5

      1  1  1  1
      2  2  2  1
B  =  1  1  1  1
      4  1  2  1
      1  1  1  1
```

For an example of a FORALL construct containing a WHERE construct with an ELSEWHERE statement, see C.4.5.

- 7 Execution of a *forall-stmt* or *forall-construct* causes the evaluation of the *subscript* and *stride* expressions
- 8 in the *forall-triplet-spec-list* for all active combinations of the *index-name* values of the outer FORALL
- 9 construct. The set of combinations of *index-name* values for the inner FORALL is the union of the
- 10 sets defined by these bounds and strides for each active combination of the outer *index-name* values; it
- 11 also includes the outer *index-name* values. The *scalar-mask-expr* is then evaluated for all combinations
- 12 of the *index-name* values of the inner construct to produce a set of active combinations for the inner
- 13 construct. If there is no *scalar-mask-expr*, it is as if it appeared with the value true. Each statement in

- 1 the inner FORALL is then executed for each active combination of the *index-name* values.

NOTE 7.60

This FORALL construct contains a nested FORALL construct. It assigns the transpose of the strict lower triangle of array A (the section below the main diagonal) to the strict upper triangle of A.

```

INTEGER A (3, 3)
FORALL (I = 1:N-1 )
  FORALL ( J=I+1:N )
    A(I,J) = A(J,I)
  END FORALL
END FORALL

```

If prior to execution N = 3 and

```

      0  3  6
A  =  1  4  7
      2  5  8

```

then after execution

```

      0  1  2
A  =  1  4  5
      2  5  8

```

2 7.2.4.3 The FORALL statement

- 3 The FORALL statement allows a single assignment statement or pointer assignment to be controlled by
4 a set of index values and an optional mask expression.

5 R759 *forall-stmt* is FORALL *forall-header forall-assignment-stmt*

- 6 A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct*
7 that is a *forall-assignment-stmt*.

- 8 The scope of an *index-name* in a *forall-stmt* is the statement itself (16.4).

NOTE 7.61

Examples of FORALL statements are:

```
FORALL (I=1:N) A(I,I) = X(I)
```

This statement assigns the elements of vector X to the elements of the main diagonal of matrix A.

```
FORALL (I = 1:N, J = 1:N) X(I,J) = 1.0 / REAL (I+J-1)
```

Array element X(I,J) is assigned the value (1.0 / REAL (I+J-1)) for values of I and J between 1 and N, inclusive.

```
FORALL (I=1:N, J=1:N, Y(I,J) /= 0 .AND. I /= J) X(I,J) = 1.0 / Y(I,J)
```

This statement takes the reciprocal of each nonzero off-diagonal element of array Y(1:N, 1:N)

NOTE 7.61 (cont.)

and assigns it to the corresponding element of array X. Elements of Y that are zero or on the diagonal do not participate, and no assignments are made to the corresponding elements of X. The results from the execution of the example in Note 7.60 could be obtained with a single FORALL statement:

```
FORALL ( I = 1:N-1, J=1:N, J > I ) A(I,J) = A(J,I)
```

For more examples of FORALL statements, see C.4.6.

1 7.2.4.4 Restrictions on FORALL constructs and statements

- 2 A many-to-one assignment is more than one assignment to the same object, or association of more
3 than one target with the same pointer, whether the object is referenced directly or indirectly through a
4 pointer. A many-to-one assignment shall not occur within a single statement in a FORALL construct or
5 statement. It is possible to assign or pointer assign to the same object in different assignment statements
6 in a FORALL construct.

NOTE 7.62

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.

- 7 Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall
8 not have the same *index-name*. The *forall-header* expressions within a nested FORALL may depend on
9 the values of outer *index-name* variables.

1 8 Execution control

2 8.1 Executable constructs containing blocks

3 8.1.1 General

4 The following are executable constructs that contain blocks:

- 5 (1) ASSOCIATE construct;
- 6 (2) BLOCK construct;
- 7 (3) CASE construct;
- 8 (4) CRITICAL construct;
- 9 (5) DO construct;
- 10 (6) IF construct;
- 11 (7) SELECT TYPE construct.

12 There is also a nonblock form of the DO construct.

13 A **block** is a sequence of executable constructs that is treated as a unit.

14 R801 *block* **is** [*execution-part-construct*] ...

15 Executable constructs may be used to control which blocks of a program are executed or how many times
 16 a block is executed. Blocks are always bounded by statements that are particular to the construct in
 17 which they are embedded; however, in some forms of the DO construct, a sequence of executable constructs without
 18 a terminating boundary statement shall obey all other rules governing blocks (8.1.2).

NOTE 8.1

A block need not contain any executable constructs. Execution of such a block has no effect.

NOTE 8.2

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
```

19 8.1.2 Rules governing blocks

20 8.1.2.1 Control flow in blocks

21 Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a
 22 block and transfers from the interior of a block to outside the block may occur.

23 Subroutine and function references (12.5.3, 12.5.4) may appear in a block.

24 8.1.2.2 Execution of a block

- 1 Execution of a block begins with the execution of the first executable construct in the block. Execution
 2 of the block is completed when the last executable construct in the sequence is executed, when a branch
 3 (8.2) within the block that has a branch target outside the block occurs, when a RETURN statement
 4 within the block is executed, or when an EXIT or CYCLE statement that belongs to a construct that
 5 contains the block is executed.

NOTE 8.3

The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct. It is usually a transfer of control.

6 8.1.3 ASSOCIATE construct

- 7 The **ASSOCIATE construct** associates named entities with expressions or variables during the exe-
 8 cution of its block. These named construct entities (16.4) are associating entities (16.5.1.6). The names
 9 are **associate names**.

10 8.1.3.1 Form of the ASSOCIATE construct

11 R802	<i>associate-construct</i>	is	<i>associate-stmt</i> <i>block</i> <i>end-associate-stmt</i>
14 R803	<i>associate-stmt</i>	is	[<i>associate-construct-name</i> :] ASSOCIATE ■ ■ (<i>association-list</i>)
16 R804	<i>association</i>	is	<i>associate-name</i> => <i>selector</i>
17 R805	<i>selector</i>	is	<i>expr</i> or <i>variable</i>
- 19 C801 (R804) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name*
 20 shall not appear in a variable definition context (16.6.7).
- 21 C802 (R804) An *associate-name* shall not be the same as another *associate-name* in the same *associate-*
 22 *stmt*.
- 23 C803 (R805) *expr* shall not be a reference to a function that has a pointer result.
- | | | | |
|---------|---------------------------|-----------|---|
| 24 R806 | <i>end-associate-stmt</i> | is | END ASSOCIATE [<i>associate-construct-name</i>] |
|---------|---------------------------|-----------|---|
- 25 C804 (R806) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*,
 26 the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the
 27 *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the cor-
 28 responding *end-associate-stmt* shall not specify an *associate-construct-name*.

29 8.1.3.2 Execution of the ASSOCIATE construct

- 30 Execution of an ASSOCIATE construct causes execution of its *associate-stmt* followed by execution
 31 of its block. During execution of that block each associate name identifies an entity, which is associ-
 32 ated (16.5.1.6) with the corresponding selector. The associating entity assumes the declared type and
 33 type parameters of the selector. If and only if the selector is polymorphic, the associating entity is
 34 polymorphic.
- 35 The other attributes of the associating entity are described in 8.1.3.3.
- 36 It is permissible to branch to an *end-associate-stmt* only from within its ASSOCIATE construct.

1 8.1.3.3 Attributes of associate names

2 Within a SELECT TYPE or ASSOCIATE construct, each associating entity has the same rank as its
 3 associated selector. The lower bound of each dimension is the result of the intrinsic function LBOUND
 4 (13.7.97) applied to the corresponding dimension of *selector*. The upper bound of each dimension is one
 5 less than the sum of the lower bound and the extent. The associating entity has the ASYNCHRONOUS
 6 or VOLATILE attribute if and only if the selector is a variable and has the attribute. The associating
 7 entity has the TARGET attribute if and only if the selector is a variable and has either the TARGET
 8 or POINTER attribute. If the associating entity is polymorphic, it assumes the dynamic type and type
 9 parameter values of the selector. If the selector has the OPTIONAL attribute, it shall be present. The
 10 associating entity is contiguous if and only if the selector is contiguous.

11 If the selector (8.1.3.1) is not permitted to appear in a variable definition context (16.6.7), the associate
 12 name shall not appear in a variable definition context.

13 8.1.3.4 Examples of the ASSOCIATE construct

NOTE 8.4

The following example illustrates an association with an expression.

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

The following example illustrates an association with a derived-type variable.

```
ASSOCIATE ( XC => AX%B(I,J)%C )
  XC%DV = XC%DV + PRODUCT(XC%EV(1:N))
END ASSOCIATE
```

The following example illustrates association with an array section.

```
ASSOCIATE ( ARRAY => AX%B(I,:)%C )
  ARRAY(N)%EV = ARRAY(N-1)%EV
END ASSOCIATE
```

The following example illustrates multiple associations.

```
ASSOCIATE ( W => RESULT(I,J)%W, ZX => AX%B(I,J)%D, ZY => AY%B(I,J)%D )
  W = ZX*X + ZY*Y
END ASSOCIATE
```

14 8.1.4 BLOCK construct

15 The **BLOCK construct** is an executable construct which may contain declarations.

16	R807	<i>block-construct</i>	is	<i>block-stmt</i>
17				[<i>specification-part</i>]
18				<i>block</i>
19				<i>end-block-stmt</i>
20	R808	<i>block-stmt</i>	is	[<i>block-construct-name</i> :] BLOCK
21	R809	<i>end-block-stmt</i>	is	END BLOCK [<i>block-construct-name</i>]

1	C805	(R807) The <i>specification-part</i> of a BLOCK construct shall not contain a COMMON, EQUIVA-
2		LENCE, IMPLICIT, INTENT, NAMELIST, OPTIONAL, or USE statement.
3	C806	(R807) A SAVE statement in a BLOCK construct shall contain a <i>saved-entity-list</i> that does not
4		specify a <i>common-block-name</i> .

J3 internal note

Unresolved Technical Issue 084

It would be better to allow the `USE` statement. Some work is required to handle modules going out of scope on exiting a `BLOCK` construct.

- 5 C807 (R807) If the *block-stmt* of a *block-construct* specifies a *block-construct-name*, the corresponding
6 *end-block-stmt* shall specify the same *block-construct-name*. If the *block-stmt* does not specify
7 a *block-construct-name*, the corresponding *end-block-stmt* shall not specify a *block-construct-*
8 *name*.

9 Except for the **ASYNCHRONOUS** and **VOLATILE** statements, specifications in a **BLOCK** construct
10 declare construct entities whose scope is that of the **BLOCK** construct (16.4).

11 Execution of a BLOCK construct causes evaluation of the specification expressions within its specification
12 part in a processor-dependent order, followed by execution of its block.

13 8.1.5 CASE construct

14 8.1.5.1 Form of the CASE construct

15 The **CASE construct** selects for execution at most one of its constituent blocks. The selection is based
16 on the value of an expression.

```

17 R810   case-construct                               is select-case-stmt
18                                     [ case-stmt
19                                       block ] ...
20                                     end-select-stmt

```

21 R811 *select-case-stmt* is [*case-construct-name* :] SELECT CASE (*case-expr*)

22 R812 *case-stmt* is CASE *case-selector* [*case-construct-name*]

23	R813	<i>end-select-stmt</i>	is	END SELECT [<i>case-construct-name</i>]

24 C808 (R810) If the *select-case-stmt* of a *case-construct* specifies a *case-construct-name*, the corre-
25 sponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt*
26 of a *case-construct* does not specify a *case-construct-name*, the corresponding *end-select-stmt*
27 shall not specify a *case-construct-name*. If a *case-stmt* specifies a *case-construct-name*, the
28 corresponding *select-case-stmt* shall specify the same *case-construct-name*.

```

29 R814    case-expr          is scalar-int-expr
30                                     or scalar-char-expr
31                                     or scalar-logical-expr

```

```

32 R815  case-selector           is ( case-value-range-list )
33                                     or DEFAULT

```

34 C809 (R810) No more than one of the selectors of one of the CASE statements shall be DEFAULT.

R816 *case-value-range* **is** *case-value*

or *case-value :*

1		or : <i>case-value</i>
2		or <i>case-value</i> : <i>case-value</i>
3	R817	is <i>scalar-int-initialization-expr</i>
4		or <i>scalar-char-initialization-expr</i>
5		or <i>scalar-logical-initialization-expr</i>

6 C810 (R810) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For
7 character type, the kind type parameters shall be the same; character length differences are
8 allowed.

9 C811 (R810) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

10 C812 (R810) For a given *case-construct*, the *case-value-ranges* shall not overlap; that is, there shall
11 be no possible value of the *case-expr* that matches more than one *case-value-range*.

12 8.1.5.2 Execution of a CASE construct

13 The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting
14 value is called the **case index**. For a case value range list, a match occurs if the case index matches any
15 of the case value ranges in the list. For a case index with a value of c , a match is determined as follows.

- 16 (1) If the case value range contains a single value v without a colon, a match occurs for type
17 logical if the expression $c \text{ .EQV. } v$ is true, and a match occurs for type integer or character
18 if the expression $c == v$ is true.
- 19 (2) If the case value range is of the form $low : high$, a match occurs if the expression $low \leq c$
20 $\text{ .AND. } c \leq high$ is true.
- 21 (3) If the case value range is of the form $low :$, a match occurs if the expression $low \leq c$ is true.
- 22 (4) If the case value range is of the form $: high$, a match occurs if the expression $c \leq high$ is
23 true.
- 24 (5) If no other selector matches and a DEFAULT selector appears, it matches the case index.
- 25 (6) If no other selector matches and the DEFAULT selector does not appear, there is no match.

26 The block following the CASE statement containing the matching selector, if any, is executed. This
27 completes execution of the construct.

28 It is permissible to branch to an *end-select-stmt* only from within its CASE construct.

29 8.1.5.3 Examples of CASE constructs

NOTE 8.5

An integer signum function:

```

INTEGER FUNCTION SIGNUM (N)
SELECT CASE (N)
CASE (:-1)
    SIGNUM = -1
CASE (0)
    SIGNUM = 0
CASE (1:)
    SIGNUM = 1
END SELECT
END

```

NOTE 8.6

A code fragment to check for balanced parentheses:

```
CHARACTER (80) :: LINE
...
LEVEL = 0
SCAN_LINE: DO I = 1, 80
    CHECK_PARENS: SELECT CASE (LINE (I:I))
        CASE ('(')
            LEVEL = LEVEL + 1
        CASE (')')
            LEVEL = LEVEL - 1
            IF (LEVEL < 0) THEN
                PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
                EXIT SCAN_LINE
            END IF
        CASE DEFAULT
            ! Ignore all other characters
    END SELECT CHECK_PARENS
END DO SCAN_LINE
IF (LEVEL > 0) THEN
    PRINT *, 'MISSING RIGHT PARENTHESIS'
END IF
```

NOTE 8.7

The following three fragments are equivalent:

```
IF (SILLY == 1) THEN
    CALL THIS
ELSE
    CALL THAT
END IF
SELECT CASE (SILLY == 1)
CASE (.TRUE.)
    CALL THIS
CASE (.FALSE.)
    CALL THAT
END SELECT
SELECT CASE (SILLY)
CASE DEFAULT
    CALL THAT
CASE (1)
    CALL THIS
END SELECT
```

NOTE 8.8

A code fragment showing several selections of one block:

```
SELECT CASE (N)
CASE (1, 3:5, 8) ! Selects 1, 3, 4, 5, 8
    CALL SUB
CASE DEFAULT
```


NOTE 8.8 (cont.)

```
CALL OTHER
END SELECT
```

1 8.1.6 CRITICAL construct

2 A **CRITICAL construct** limits execution of a block to one image at a time.

```
3 R818  critical-construct      is  critical-stmt
4                                     block
5                                     end-critical-stmt
```

```
6 R819  critical-stmt          is  [ critical-construct-name : ] CRITICAL
```

```
7 R820  end-critical-stmt      is  END CRITICAL [ critical-construct-name ]
```

8 C813 (R818) If the *critical-stmt* of a *critical-construct* specifies a *critical-construct-name*, the corre-
 9 sponding *end-critical-stmt* shall specify the same *critical-construct-name*. If the *critical-stmt* of a
 10 *critical-construct* does not specify a *critical-construct-name*, the corresponding *end-critical-stmt*
 11 shall not specify a *critical-construct-name*.

12 C814 (R818) The *block* of a *critical-construct* shall not contain an image control statement.

13 Execution of the **CRITICAL** construct is completed when execution of its block is completed.

14 The processor shall ensure that once an image has commenced executing *block*, no other image shall
 15 commence executing it until the image has completed executing it. If image T is the next to execute the
 16 construct after image M, the segments (8.5.1) that executed before the construct on image M precede
 17 the segments that execute after the construct on image T. No image control statement shall be executed
 18 during the execution of a critical construct by the image executing the **CRITICAL** construct.

NOTE 8.9

If more than one image executes the block of a **CRITICAL** construct, its execution by one image always either precedes or succeeds its execution by another image. Typically no other statement ordering is needed. Consider the following example:

```
CRITICAL
  GLOBAL_COUNTER[1] = GLOBAL_COUNTER[1] + 1
END CRITICAL
```

The definition of GLOBAL_COUNTER[1] by a particular image will always precede the reference to the same variable by the next image to execute the block.

NOTE 8.10

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]
```

NOTE 8.10 (cont.)

```

    NUM_JOBS[1] = JOB - 1
  END CRITICAL
  IF (JOB > 0) THEN
    ! Work on JOB
  ELSE
    EXIT
  END IF
END DO
SYNC ALL

```

1 8.1.7 DO construct**2 8.1.7.1 General**

3 The **DO construct** specifies the repeated execution of a sequence of executable constructs. Such a
4 repeated sequence is called a **loop**.

5 The number of iterations of a loop may be determined at the beginning of execution of the DO construct,
6 or may be left indefinite (“DO forever” or DO WHILE). Except in the case of a DO CONCURRENT
7 construct, the loop can be terminated immediately (8.1.7.5.4). The current iteration of the loop may be
8 curtailed by executing a CYCLE statement (8.1.7.5.3).

9 There are three phases in the execution of a DO construct: initiation of the loop, execution of the loop
10 range, and termination of the loop.

11 The **DO CONCURRENT construct** is a DO construct whose DO statement contains the CON-
12 CURRENT keyword.

13 8.1.7.2 Forms of the DO construct

14 The DO construct may be written in either a block form or a nonblock form.

15 R821 *do-construct* **is** *block-do-construct*
16 **or** *nonblock-do-construct*

17 8.1.7.2.1 Form of the block DO construct

18 R822 *block-do-construct* **is** *do-stmt*
19 *do-block*
20 *end-do*

21 R823 *do-stmt* **is** *label-do-stmt*
22 **or** *nonlabel-do-stmt*

23 R824 *label-do-stmt* **is** [*do-construct-name* :] DO *label* [*loop-control*]

24 R825 *nonlabel-do-stmt* **is** [*do-construct-name* :] DO [*loop-control*]

25 R826 *loop-control* **is** [,] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■
26 ■ [, *scalar-int-expr*]
27 **or** [,] WHILE (*scalar-logical-expr*)
28 **or** [,] CONCURRENT *forall-header*

29 R827 *do-variable* **is** *scalar-int-variable-name*

1 C815 (R827) The *do-variable* shall be a variable of type integer.

2 R828 *do-block* **is** *block*

3 R829 *end-do* **is** *end-do-stmt*
4 **or** *continue-stmt*

5 R830 *end-do-stmt* **is** END DO [*do-construct-name*]

6 C816 (R822) If the *do-stmt* of a *block-do-construct* specifies a *do-construct-name*, the corresponding
7 *end-do* shall be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a
8 *block-do-construct* does not specify a *do-construct-name*, the corresponding *end-do* shall not
9 specify a *do-construct-name*.

10 C817 (R822) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

11 C818 (R822) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the
12 same *label*.

13 8.1.7.2.2 Form of the nonblock DO construct

14 R831 *nonblock-do-construct* **is** *action-term-do-construct*
15 **or** *outer-shared-do-construct*

16 R832 *action-term-do-construct* **is** *label-do-stmt*
17 *do-body*
18 *do-term-action-stmt*

19 R833 *do-body* **is** [*execution-part-construct*] ...

20 R834 *do-term-action-stmt* **is** *action-stmt*

21 C819 (R834) A *do-term-action-stmt* shall not be an *arithmetic-if-stmt*, *continue-stmt*, *cycle-stmt*, *end-function-stmt*,
22 *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, *exit-stmt*, *goto-stmt*, *return-stmt*, or *stop-*
23 *stmt*.

24 C820 (R831) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer
25 to the same label.

26 R835 *outer-shared-do-construct* **is** *label-do-stmt*
27 *do-body*
28 *shared-term-do-construct*

29 R836 *shared-term-do-construct* **is** *outer-shared-do-construct*
30 **or** *inner-shared-do-construct*

31 R837 *inner-shared-do-construct* **is** *label-do-stmt*
32 *do-body*
33 *do-term-shared-stmt*

34 R838 *do-term-shared-stmt* **is** *action-stmt*

35 C821 (R838) A *do-term-shared-stmt* shall not be an *arithmetic-if-stmt*, *cycle-stmt*, *end-function-stmt*, *end-program-*
36 *stmt*, *end-mp-subprogram-stmt*, *end-subroutine-stmt*, *exit-stmt*, *goto-stmt*, *return-stmt*, or *stop-stmt*.

37 C822 (R836) The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmts* of the *inner-shared-*
38 *do-construct* and *outer-shared-do-construct* shall refer to the same label.

39 The *do-term-action-stmt*, *do-term-shared-stmt*, or *shared-term-do-construct* following the *do-body* of a nonblock DO con-
40 struct is called the **DO termination** of that construct.

41 Within a scoping unit, all DO constructs whose DO statements refer to the same label are nonblock DO constructs, and
42 share the statement identified by that label.

1 8.1.7.3 Range of the DO construct

2 The **range** of a block DO construct is the *do-block*, which shall satisfy the rules for blocks (8.1.2). In
3 particular, transfer of control to the interior of such a block from outside the block is prohibited. It
4 is permitted to branch to the *end-do* of a block DO construct only from within the range of that DO
5 construct.

6 The range of a nonblock DO construct consists of the *do-body* and the following DO termination. The end of such a
7 range is not bounded by a particular statement as for the other executable constructs (e.g., END IF); nevertheless, the
8 range satisfies the rules for blocks (8.1.2). Transfer of control into the *do-body* or to the DO termination from outside the
9 range is prohibited; in particular, it is permitted to branch to a *do-term-shared-stmt* only from within the range of the
10 corresponding *inner-shared-do-construct*.

11 8.1.7.4 Active and inactive DO constructs

12 A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only
13 when its DO statement is executed.

14 Once active, the DO construct becomes inactive only when it terminates (8.1.7.5.4).

15 8.1.7.5 Execution of a DO construct

16 8.1.7.5.1 Loop initiation

17 When the DO statement is executed, the DO construct becomes active. If *loop-control* is

18 $[,] \text{ } do\text{-}variable = scalar\text{-}int\text{-}expr_1 , scalar\text{-}int\text{-}expr_2 [, scalar\text{-}int\text{-}expr_3]$

19 the following steps are performed in sequence.

- 20 (1) The initial parameter m_1 , the terminal parameter m_2 , and the incrementation parameter
21 m_3 are of type integer with the same kind type parameter as the *do-variable*. Their values
22 are established by evaluating *scalar-int-expr₁*, *scalar-int-expr₂*, and *scalar-int-expr₃*, re-
23 spectively, including, if necessary, conversion to the kind type parameter of the *do-variable*
24 according to the rules for numeric conversion (Table 7.13). If *scalar-int-expr₃* does not
25 appear, m_3 has the value 1. The value of m_3 shall not be zero.
- 26 (2) The DO variable becomes defined with the value of the initial parameter m_1 .
- 27 (3) The **iteration count** is established and is the value of the expression $(m_2 - m_1 + m_3)/m_3$,
28 unless that value is negative, in which case the iteration count is 0.

NOTE 8.11

The iteration count is zero whenever:

$$\begin{aligned} & m_1 > m_2 \text{ and } m_3 > 0, \text{ or} \\ & m_1 < m_2 \text{ and } m_3 < 0. \end{aligned}$$

29 If *loop-control* is omitted, no iteration count is calculated. The effect is as if a large positive iteration
30 count, impossible to decrement to zero, were established. If *loop-control* is $[,]$ WHILE (*scalar-logical-*
31 *expr*), the effect is as if *loop-control* were omitted and the following statement inserted as the first
32 statement of the *do-block*:

33 IF (.NOT. (*scalar-logical-expr*)) EXIT

34 For a DO CONCURRENT construct, the values of the index variables for the iterations of the construct
35 are determined by the rules for the index variables of the FORALL construct (7.2.4.2.2 and 7.2.4.2.3).

36 An *index-name* in a DO CONCURRENT construct has a scope of the construct (16.4). It is a scalar

1 variable that has the type and type parameters that it would have if it were the name of a variable in the
2 scoping unit that includes the construct, and this type shall be integer type; it has no other attributes.

3 At the completion of the execution of the DO statement, the execution cycle begins.

4 **8.1.7.5.2 The execution cycle**

5 The **execution cycle** of a DO construct that is not a DO CONCURRENT construct consists of the
6 following steps performed in sequence repeatedly until termination.

7 (1) The iteration count, if any, is tested. If it is zero, the loop terminates and the DO construct
8 becomes inactive. If *loop-control* is [,] WHILE (*scalar-logical-expr*), the *scalar-logical-expr*
9 is evaluated; if the value of this expression is false, the loop terminates and the DO construct
10 becomes inactive. If, as a result, all of the DO constructs sharing the *do-term-shared-stmt* are inactive,
11 the execution of all of these constructs is complete. However, if some of the DO constructs sharing the
12 *do-term-shared-stmt* are active, execution continues with step (3) of the execution cycle of the active DO
13 construct whose DO statement was most recently executed.

14 (2) The range of the loop is executed.

15 (3) The iteration count, if any, is decremented by one. The DO variable, if any, is incremented
16 by the value of the incrementation parameter *m*₃.

17 Except for the incrementation of the DO variable that occurs in step (3), the DO variable shall neither
18 be redefined nor become undefined while the DO construct is active.

19 The range of a DO CONCURRENT construct is executed for all of the active combinations of the
20 *index-name* values. Each execution of the range is an iteration. The executions may occur in any order.

21 **8.1.7.5.3 CYCLE statement**

22 Execution of the range of the loop may be curtailed by executing a CYCLE statement from within the
23 range of the loop.

24 R839 *cycle-stmt* **is** CYCLE [*do-construct-name*]

25 C823 (R839) If a *do-construct-name* appears, the CYCLE statement shall be within the range of that
26 *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.

27 C824 (R839) A *cycle-stmt* shall not appear within the range of a DO CONCURRENT construct if it
28 belongs to an outer construct.

29 A CYCLE statement **belongs** to a particular DO construct. If the CYCLE statement contains a DO
30 construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct
31 in which it appears.

32 Execution of a CYCLE statement that belongs to a DO construct that is not a DO CONCURRENT
33 construct causes immediate progression to step (3) of the current execution cycle of the DO construct
34 to which it belongs. If this construct is a nonblock DO construct, the *do-term-action-stmt* or *do-term-shared-stmt* is
35 not executed.

36 Execution of a CYCLE statement that belongs to a DO CONCURRENT construct completes execution
37 of that iteration of the construct.

38 In a block DO construct, a transfer of control to the *end-do* has the same effect as execution of a CYCLE
39 statement belonging to that construct. In a nonblock DO construct, transfer of control to the *do-term-action-stmt*
40 or *do-term-shared-stmt* causes that statement to be executed. Unless a further transfer of control results, step (3) of the
41 current execution cycle of the DO construct is then executed.

1 8.1.7.5.4 Loop termination

2 For a DO construct that is not a DO CONCURRENT construct, the loop terminates, and the DO
3 construct becomes inactive, when any of the following occurs.

- 4 (1) The iteration count is determined to be zero or the *scalar-logical-expr* is false, when tested
5 during step (1) of the above execution cycle.
- 6 (2) An EXIT statement belonging to the DO construct is executed.
- 7 (3) An EXIT or CYCLE statement that belongs to an outer construct and is within the range
8 of the DO construct is executed.
- 9 (4) Control is transferred from a statement within the range of a DO construct to a statement
10 that is neither the *end-do* nor within the range of the same DO construct.
- 11 (5) A RETURN statement within the range of the DO construct is executed.

12 For a DO CONCURRENT construct, the loop terminates, and the DO construct becomes inactive when
13 all of the iterations have completed execution.

14 When a DO construct becomes inactive, the DO variable, if any, of the DO construct retains its last
15 defined value.

16 8.1.7.6 Restrictions on DO CONCURRENT constructs

17 C825 A RETURN statement shall not appear within a DO CONCURRENT construct.

18 C826 A branch (8.2) within a DO CONCURRENT construct shall not have a branch target that is
19 outside the construct.

20 C827 A reference to a nonpure procedure shall not appear within a DO CONCURRENT construct.

21 C828 A reference to the procedure IEEE.GET_FLAG, IEEE.SET_HALTING_MODE, or IEEE.GET_-
22 HALTING_MODE from the intrinsic module IEEE.EXCEPTIONS, shall not appear within a
23 DO CONCURRENT construct.

24 The following additional restrictions apply to DO CONCURRENT constructs.

- 25 • A variable that is referenced in an iteration shall either be previously defined during that iteration,
26 or shall not be defined or become undefined during any other iteration of the current execution of
27 the construct. A variable that is defined or becomes undefined by more than one iteration of the
28 current execution of the construct becomes undefined when the current execution of the construct
29 terminates.
- 30 • A pointer that is referenced in an iteration either shall be previously pointer associated during that
31 iteration, or shall not have its pointer association changed during any iteration. A pointer that has
32 its pointer association changed in more than one iteration has a processor dependent association
33 status when the construct terminates.
- 34 • An allocatable object that is allocated in more than one iteration shall be subsequently deallocated
35 during the same iteration in which it was allocated. An object that is allocated or deallocated in
36 only one iteration shall not be deallocated, allocated, referenced, defined, or become undefined in
37 a different iteration.
- 38 • An input/output statement shall not write data to a file record or position in one iteration and
39 read from the same record or position in a different iteration of the same execution of the construct.
- 40 • Records written by output statements in the loop range to a sequential access file appear in the
41 file in an indeterminate order.

NOTE 8.12

The restrictions on referencing variables defined in an iteration of a DO CONCURRENT construct apply to any procedure invoked within the loop.

NOTE 8.13

The restrictions on the statements in the loop range of 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 characteristics of the program not visible to the compiler.

1 8.1.7.7 Examples of DO constructs

NOTE 8.14

The following program fragment computes a tensor product of two arrays:

```
DO I = 1, M
  DO J = 1, N
    C (I, J) = SUM (A (I, J, :) * B (:, I, J))
  END DO
END DO
```

NOTE 8.15

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 range of the loop.

```
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
DO WHILE (IOS == 0)
  IF (X >= 0.) THEN
    CALL SUBA (X)
    CALL SUBB (X)
    ...
    CALL SUBZ (X)
  ENDIF
  READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO
```

NOTE 8.16

The following example behaves exactly the same as the one in Note 8.15. However, the READ statement has been moved to the interior of the range, 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)
```

NOTE 8.16 (cont.)

```

      . . .
      CALL SUBZ (X)
END DO

```

NOTE 8.17

The following example represents a case in which the user knows that the elements of the array IND form a permutation of the integers 1, ..., N. 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,N),IND(N)

DO CONCURRENT (I=1:N, J=1:N)
  A(IND(I),IND(J)) = A(IND(I),IND(J)) + 1
END DO

```

NOTE 8.18

Additional examples of DO constructs are in C.5.3.

1 8.1.8 IF construct and statement**2 8.1.8.1 Form of the IF construct**

3 The **IF construct** selects for execution at most one of its constituent blocks. The selection is based on
4 a sequence of logical expressions.

```

5 R840  if-construct           is  if-then-stmt
6                                     block
7                                     [ else-if-stmt
8                                     block ] ...
9                                     [ else-stmt
10                                    block ]
11                                    end-if-stmt

12 R841  if-then-stmt           is  [ if-construct-name : ] IF ( scalar-logical-expr ) THEN
13 R842  else-if-stmt           is  ELSE IF ( scalar-logical-expr ) THEN [ if-construct-name ]
14 R843  else-stmt              is  ELSE [ if-construct-name ]
15 R844  end-if-stmt            is  END IF [ if-construct-name ]

```

16 C829 (R840) If the *if-then-stmt* of an *if-construct* specifies an *if-construct-name*, the corresponding
17 *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* does
18 not specify an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-*
19 *name*. If an *else-if-stmt* or *else-stmt* specifies an *if-construct-name*, the corresponding *if-then-*
20 *stmt* shall specify the same *if-construct-name*.

21 8.1.8.2 Execution of an IF construct

22 At most one of the blocks in the IF construct is executed. If there is an ELSE statement in the construct,
23 exactly one of the blocks in the construct is executed. The scalar logical expressions are evaluated in

- 1 the order of their appearance in the construct until a true value is found or an ELSE statement or END
 2 IF statement is encountered. If a true value or an ELSE statement is found, the block immediately
 3 following is executed and this completes the execution of the construct. The scalar logical expressions
 4 in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated
 5 expressions is true and there is no ELSE statement, the execution of the construct is completed without
 6 the execution of any block within the construct.
- 7 It is permissible to branch to an END IF statement only from within its IF construct. Execution of an
 8 END IF statement has no effect.

9 8.1.8.3 Examples of IF constructs

NOTE 8.19

```

IF (CVAR == 'RESET') THEN
  I = 0; J = 0; K = 0
END IF
PROOF_DONE: IF (PROP) THEN
  WRITE (3, '( 'QED' )')
  STOP
ELSE
  PROP = NEXTPROP
END IF PROOF_DONE
IF (A > 0) THEN
  B = C/A
  IF (B > 0) THEN
    D = 1.0
  END IF
ELSE IF (C > 0) THEN
  B = A/C
  D = -1.0
ELSE
  B = ABS (MAX (A, C))
  D = 0
END IF

```

10 8.1.8.4 IF statement

- 11 The **IF statement** controls the execution of a single action statement based on a single logical expression.
- 12 R845 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*
- 13 C830 (R845) The *action-stmt* in the *if-stmt* shall not be an *end-function-stmt*, *end-mp-subprogram-*
 14 *stmt*, *end-program-stmt*, *end-subroutine-stmt*, or *if-stmt*.
- 15 Execution of an IF statement causes evaluation of the scalar logical expression. If the value of the
 16 expression is true, the action statement is executed. If the value is false, the action statement is not
 17 executed and execution continues.
- 18 The execution of a function reference in the scalar logical expression may affect entities in the action
 19 statement.

NOTE 8.20

An example of an IF statement is:

```
IF ( A > 0.0 ) A = LOG ( A )
```

1 8.1.9 SELECT TYPE construct

2 8.1.9.1 Form of the SELECT TYPE construct

3 The **SELECT TYPE construct** selects for execution at most one of its constituent blocks. The
4 selection is based on the dynamic type of an expression. A name is associated with the expression or
5 variable (16.4, 16.5.1.6), in the same way as for the ASSOCIATE construct.

```
6 R846  select-type-construct      is  select-type-stmt
7                                     [ type-guard-stmt
8                                     block ] ...
9                                     end-select-type-stmt
```

```
10 R847  select-type-stmt          is  [ select-construct-name : ] SELECT TYPE ■
11                                     ■ ( [ associate-name => ] selector )
```

12 C831 (R847) If *selector* is not a named *variable*, *associate-name =>* shall appear.

13 C832 (R847) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name*
14 shall not appear in a variable definition context (16.6.7).

15 C833 (R847) The *selector* in a *select-type-stmt* shall be polymorphic.

```
16 R848  type-guard-stmt           is  TYPE IS ( type-spec ) [ select-construct-name ]
17                                     or  CLASS IS ( derived-type-spec ) [ select-construct-name ]
18                                     or  CLASS DEFAULT [ select-construct-name ]
```

19 C834 (R848) The *type-spec* or *derived-type-spec* shall specify that each length type parameter is as-
20 sumed.

21 C835 (R848) The *type-spec* or *derived-type-spec* shall not specify a sequence derived type or a type
22 with the BIND attribute.

23 C836 (R848) If *selector* is not unlimited polymorphic, the *type-spec* or *derived-type-spec* shall specify
24 an extension of the declared type of *selector*.

25 C837 (R848) For a given *select-type-construct*, the same type and kind type parameter values shall
26 not be specified in more than one TYPE IS *type-guard-stmt* and shall not be specified in more
27 than one CLASS IS *type-guard-stmt*.

28 C838 (R848) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-*
29 *guard-stmt*.

```
30 R849  end-select-type-stmt      is  END SELECT [ select-construct-name ]
```

31 C839 (R846) If the *select-type-stmt* of a *select-type-construct* specifies a *select-construct-name*, the
32 corresponding *end-select-type-stmt* shall specify the same *select-construct-name*. If the *select-*
33 *type-stmt* of a *select-type-construct* does not specify a *select-construct-name*, the corresponding
34 *end-select-type-stmt* shall not specify a *select-construct-name*. If a *type-guard-stmt* specifies a
35 *select-construct-name*, the corresponding *select-type-stmt* shall specify the same *select-construct-*
36 *name*.

1 The associate name of a SELECT TYPE construct is the *associate-name* if specified; otherwise it is the
 2 *name* that constitutes the *selector*.

3 8.1.9.2 Execution of the SELECT TYPE construct

4 Execution of a SELECT TYPE construct whose selector is not a *variable* causes the selector expression
 5 to be evaluated.

6 A SELECT TYPE construct selects at most one block to be executed. During execution of that block,
 7 the associate name identifies an entity, which is associated (16.5.1.6) with the selector.

8 A TYPE IS type guard statement matches the selector if the dynamic type and kind type parameter
 9 values of the selector are the same as those specified by the statement. A CLASS IS type guard
 10 statement matches the selector if the dynamic type of the selector is an extension of the type specified
 11 by the statement and the kind type parameter values specified by the statement are the same as the
 12 corresponding type parameter values of the dynamic type of the selector.

13 The block to be executed is selected as follows.

- 14 (1) If a TYPE IS type guard statement matches the selector, the block following that statement
 15 is executed.
- 16 (2) Otherwise, if exactly one CLASS IS type guard statement matches the selector, the block
 17 following that statement is executed.
- 18 (3) Otherwise, if several CLASS IS type guard statements match the selector, one of these
 19 statements must specify a type that is an extension of all the types specified in the others;
 20 the block following that statement is executed.
- 21 (4) Otherwise, if there is a CLASS DEFAULT type guard statement, the block following that
 22 statement is executed.

NOTE 8.21

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.

23 Within the block following a TYPE IS type guard statement, the associating entity (16.5.5) is not
 24 polymorphic (4.3.1.3), has the type named in the type guard statement, and has the type parameter
 25 values of the selector.

26 Within the block following a CLASS IS type guard statement, the associating entity is polymorphic and
 27 has the declared type named in the type guard statement. The type parameter values of the associating
 28 entity are the corresponding type parameter values of the selector.

29 Within the block following a CLASS DEFAULT type guard statement, the associating entity is poly-
 30 morphic and has the same declared type as the selector. The type parameter values of the associating
 31 entity are those of the declared type of the selector.

NOTE 8.22

If the declared type of the *selector* is T, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (T).

32 The other attributes of the associating entity are described in 8.1.3.3.

33 It is permissible to branch to an *end-select-type-stmt* only from within its SELECT TYPE construct.

34 8.1.9.3 Examples of the SELECT TYPE construct

NOTE 8.23

```

TYPE POINT
  REAL :: X, Y
END TYPE POINT
TYPE, EXTENDS(POINT) :: POINT_3D
  REAL :: Z
END TYPE POINT_3D
TYPE, EXTENDS(POINT) :: COLOR_POINT
  INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT), TARGET :: P
TYPE(POINT_3D), TARGET :: P3
TYPE(COLOR_POINT), TARGET :: C
CLASS(POINT), POINTER :: P_OR_C
P_OR_C => C
SELECT TYPE ( A => P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: A" implied here
  PRINT *, A%X, A%Y ! This block gets executed
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: A" implied here
  PRINT *, A%X, A%Y, A%Z
END SELECT

```

NOTE 8.24

The following example illustrates the omission of *associate-name*. It uses the declarations from Note 8.23.

```

P_OR_C => P3
SELECT TYPE ( P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y, P_OR_C%Z ! This block gets executed
END SELECT

```

1 **8.1.10 EXIT statement**

2 The **EXIT statement** provides one way of terminating a construct.

3 R850 *exit-stmt* **is** EXIT [*construct-name*]

4 C840 If a *construct-name* appears, the EXIT statement shall be within that construct; otherwise, it
5 shall be within the range of at least one *do-construct*.

6 An EXIT statement **belongs** to a particular construct. If the EXIT statement contains a construct
7 name, it belongs to that construct; otherwise, it belongs to the innermost DO construct in which it
8 appears.

1 C841 An *exit-stmt* shall not belong to a DO CONCURRENT construct, nor shall it appear within
 2 the range of a DO CONCURRENT construct if it belongs to a construct that contains that DO
 3 CONCURRENT construct.

4 When an EXIT statement that belongs to a DO construct is executed, it terminates the loop (8.1.7.5.4)
 5 and any active loops contained within the terminated loop. When an EXIT statement that belongs to
 6 a non-DO construct is executed, it terminates any active loops contained within that construct, and
 7 completes execution of that construct.

8 8.2 Branching

9 **Branching** is used to alter the normal execution sequence. A branch causes a transfer of control from
 10 one statement in a scoping unit to a labeled branch target statement in the same scoping unit. Branching
 11 may be caused by a GOTO statement, a computed GOTO statement, an arithmetic IF statement, a
 12 CALL statement that has an *alt-return-spec*, or an input/output statement that has an END= or ERR=
 13 specifier. Although procedure references and control constructs can cause transfer of control, they are
 14 not branches. A **branch target statement** is an *action-stmt*, an *associate-stmt*, an *end-associate-*
 15 *stmt*, an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, an *end-select-stmt*, a *select-type-stmt*, an *end-*
 16 *select-type-stmt*, a *do-stmt*, an *end-do-stmt*, *block-stmt*, *end-block-stmt*, *critical-stmt*, *end-critical-stmt*,
 17 a *forall-construct-stmt*, a *do-term-action-stmt*, a *do-term-shared-stmt*, or a *where-construct-stmt*.

18 8.2.1 GO TO statement

19 5

20 R851 *goto-stmt* is GO TO *label*

21 C842 (R851) The *label* shall be the statement label of a branch target statement that appears in the
 22 same scoping unit as the *goto-stmt*.

23 Execution of a **GO TO statement** causes a transfer of control so that the branch target statement
 24 identified by the label is executed next.

25 8.2.2 Computed GO TO statement

26 R852 *computed-goto-stmt* is GO TO (*label-list*) [,] *scalar-int-expr*

27 C843 (R852) Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same
 28 scoping unit as the *computed-goto-stmt*.

NOTE 8.25

The same statement label may appear more than once in a label list.

29 Execution of a **computed GO TO statement** causes evaluation of the scalar integer expression. If this value is *i* such
 30 that $1 \leq i \leq n$ where *n* is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed
 31 is the one identified by the *i*th label in the list of labels. If *i* is less than 1 or greater than *n*, the execution sequence
 32 continues as though a CONTINUE statement were executed.

33 8.2.3 Arithmetic IF statement

34 R853 *arithmetic-if-stmt* is IF (*scalar-numeric-expr*) *label* , *label* , *label*

35 C844 (R853) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the
 36 *arithmetic-if-stmt*.

37 C845 (R853) The *scalar-numeric-expr* shall not be of type complex.

NOTE 8.26

The same label may appear more than once in one arithmetic IF statement.

- 1 Execution of an **arithmetic IF statement** causes evaluation of the numeric expression followed by a transfer of control.
- 2 The branch target statement identified by the first label, the second label, or the third label is executed next depending
- 3 on whether the value of the numeric expression is less than zero, equal to zero, or greater than zero, respectively.

4 **8.3 CONTINUE statement**

- 5 Execution of a **CONTINUE statement** has no effect.

6 R854 *continue-stmt* **is** CONTINUE

7 **8.4 STOP statement**

8 R855 *stop-stmt* **is** STOP [*stop-code*]

9 R856 *stop-code* **is** *scalar-char-initialization-expr*
 10 **or** *scalar-int-initialization-expr*

11 C846 (R856) The *scalar-char-initialization-expr* shall be of default kind.

12 C847 (R856) The *scalar-int-initialization-expr* shall be of default kind.

- 13 Execution of a **STOP statement** causes normal termination of execution of that image. If each image
- 14 of a set of images executes a STOP statement immediately following the execution of a construct that
- 15 performs a synchronization of the images in the set, normal termination of execution occurs for all of
- 16 the images in the set; the executions of all other images are terminated. If termination of execution of
- 17 an image occurs for some other reason, termination of execution occurs on all other images.

J3 internal note

Unresolved Technical Issue 005

1. Context-dependent meaning (immediately after SYNC ALL) is a bad idea, especially since “immediately following” is not well-defined.
2. Mixing “normal termination” and “error termination” is also probably a bad idea. In the past, STOP has always been “normal”, and various other terminations have been “error” (DEALLOCATE of an unallocated allocatable). We have never felt the need to have a special statement for causing error termination, and having “STOP,ALL::” do “normal” termination on some images and “error” termination on other images is baroque.
3. We do not need to have semantics like “what happens when the user presses Control-C”; we have never had anything in the Fortran standard that did that before either.
4. Notation (editorial): we should probably define the term “error termination” and use it in the multiple places we mean that, instead of merely omitting “normal” before “termination”.

J3 internal note (cont.)**J3 internal note****Unresolved Technical Issue 006**

There is no explanation, let alone an adequate one, of when this (termination of execution of remote images) occurs. That means that there is no reason for the processor to terminate any other image, it can just let them run.

Surely we want to at least terminate a remote image at the next attempt to SYNC with the image that executes “STOP,ALL:”.

NOTE 8.27

If all images execute a SYNC ALL statement and immediately afterwards execute a STOP statement, normal termination of execution occurs on all images.

If only a subset of the images have their executions terminate normally, how soon termination takes place on the other images is processor dependent.

- 1 When an image is terminated by a STOP statement, its stop code, if any, is made available in a
- 2 processor-dependent manner. If any exception (14) is signaling on that image, the processor shall issue
- 3 a warning indicating which exceptions are signaling; this warning shall be on the unit identified by the
- 4 named constant ERROR_UNIT (13.8.2.5). It is recommended that the stop code is made available by
- 5 formatted output to the same unit.

NOTE 8.28

When normal termination occurs on more than one image, it is expected that a processor-dependent summary of the stop codes and signaling exceptions will be made available.

NOTE 8.29

If the *stop-code* is an integer, it is recommended that the value also be used as the process exit status, if the processor supports that concept. If the integer *stop-code* is used as the process exit status, the processor might be able to interpret only values within a limited range, or only a limited portion of the integer value (for example, only the least-significant 8 bits).

If the *stop-code* is of type character or does not appear, or if an END PROGRAM statement is executed, it is recommended that the value zero be supplied as the process exit status, if the processor supports that concept.

6 **8.5 Image execution control**

7 **8.5.1 Image control statements**

8 The execution sequence on each image is as specified in 2.3.6.

9 An **image control** statement affects the execution ordering between images. Each of the following is
10 an image control statement:

- 11 • SYNC ALL statement;
- 12 • SYNC TEAM statement;
- 13 • SYNC IMAGES statement;
- 14 • SYNC MEMORY statement;

- 1 • NOTIFY statement;
- 2 • QUERY statement;
- 3 • ALLOCATE or DEALLOCATE statement involving a co-array;
- 4 • CRITICAL or END CRITICAL statement (8.1.6);
- 5 • OPEN statement with a TEAM= specifier;
- 6 • CLOSE statement;
- 7 • END, END BLOCK, or RETURN statement that involves an implicit deallocation of a co-array;
- 8 • END PROGRAM statement;
- 9 • CALL statement for a collective subroutine (13.1) or the FORM_TEAM intrinsic subroutine
- 10 (13.7.69).

11 During an execution of a statement that contains more than one reference to a procedure, at most one
 12 reference shall cause execution of an image control statement other than CRITICAL, END CRITICAL,
 13 or CLOSE for a file with a connect term of only one image.

14 On each image, the sequence of statements executed before the first image control statement, between
 15 the execution of two image control statements, or after the last image control statement is known as
 16 a **segment**. The segment executed immediately before the execution of an image control statement
 17 includes the evaluation of all expressions within the statement.

18 By execution of image control statements or user-defined ordering (8.5.6), the program can ensure that
 19 the execution of the i^{th} segment on image P, P_i , either precedes or succeeds the execution of the j^{th}
 20 segment on another image Q, Q_j . If the program does not ensure this, segments P_i and Q_j are unordered;
 21 depending on the relative execution speeds of the images, some or all of the execution of the segment P_i
 22 may take place at the same time as some or all of the execution of the segment Q_j .

NOTE 8.30

The set of all segments on all images is partially ordered: the segment P_i precedes segment Q_j if and only if there is a sequence of segments starting with P_i and ending with Q_j such that each segment of the sequence precedes the next either because they are on the same image or because of the execution of image control statements.

23 A scalar co-variable that is of type default integer, default logical, default real, or default bits, and has
 24 the VOLATILE attribute (5.3.18) may be referenced during the execution of a segment that is unordered
 25 relative to the execution of a segment in which the co-variable is defined. Otherwise,

- 26 • if a co-array variable is defined on an image in a segment, it shall not be referenced, defined, or
 27 become undefined in a segment on another image unless the segments are ordered,
- 28 • if the allocation of an allocatable subobject of a co-array or the pointer association of a pointer
 29 subobject of a co-array is changed on an image in a segment, that subobject shall not be referenced
 30 or defined in a segment on another image unless the segments are ordered, and
- 31 • if a procedure invocation on image P is in execution in segments $P_i, P_{i+1}, \dots, P_k$ and defines a
 32 non-co-array dummy argument, the argument associated entity shall not be referenced, defined, or
 33 become undefined on another image Q in a segment Q_j unless Q_j precedes P_i or succeeds P_k .

NOTE 8.31

Apart from the effects of volatile variables, the processor may optimize the execution of a segment as if it were the only image in execution.

NOTE 8.32

The model upon which the interpretation of a program is based is that there is a permanent memory location for each co-array variable and that all images can access it. In practice, an image may make a copy of a non-volatile co-array variable (in cache or a register, for example) and, as an optimization, defer copying a changed value back to the permanent location while it is still being used. Since the variable is not volatile, it is safe to defer this transfer until the end of the current segment and thereafter to reload from permanent memory any co-array variable that was not defined within the segment. It would not be safe to defer these actions beyond the end of the current segment since another image might reference the variable then.

- 1 If an image P writes a record during the execution of P_i to a file that is opened for direct access with a
- 2 TEAM= specifier, no other image Q shall read or write the record during execution of a segment that
- 3 is unordered with P_i . Furthermore, it shall not read the record in a segment that succeeds P_i unless
- 4 • after image P writes the record, it executes a FLUSH statement (9.8) for the file during the
- 5 execution of a segment P_k , where $k \geq i$, and
- 6 • before image Q reads the record, it executes a FLUSH statement for the file during the execution
- 7 of a segment Q_j that succeeds P_k .

NOTE 8.33

The incorrect sequencing of image control statements can halt execution indefinitely. For example, one image might be executing a SYNC ALL statement while another is executing an ALLOCATE statement for a co-array; or one image might be executing a blocking QUERY statement for which an image in its image set never executes the corresponding NOTIFY statement.

8.5.2 SYNC ALL statement

9 R857 *sync-all-stmt* **is** SYNC ALL [([*sync-stat-list*])]

10 R858 *sync-stat* **is** STAT = *stat-variable*
 11 **or** ERRMSG = *errmsg-variable*

12 C848 No specifier shall appear more than once in a given *sync-stat-list*.

13 The STAT= and ERRMSG= specifiers for image execution control statements are described in 8.5.7.

14 Execution of a SYNC ALL statement performs a synchronization of all images. Execution on an image,
 15 M, of the segment following the SYNC ALL statement is delayed until each other image has executed a
 16 SYNC ALL statement as many times as has image M. The segments that executed before the SYNC ALL
 17 statement on an image precede the segments that execute after the SYNC ALL statement on another
 18 image.

NOTE 8.34

If synchronization is required when the images commence statement execution, a SYNC ALL statement should be the first executable statement of the main program. This is necessary if the code relies on the initialization of a co-array variable on another image.

NOTE 8.35

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

```

1 8.5.3 SYNC TEAM statement

2 R859 *sync-team-stmt* is SYNC TEAM (*image-team* [, *sync-stat-list*])

3 R860 *image-team* is *scalar-variable*

4 C849 The *image-team* shall be a scalar variable of type IMAGE_TEAM from the intrinsic module
5 ISO_FORTRAN_ENV.

6 Execution of a SYNC TEAM statement performs a **team synchronization**, which is a synchronization
7 of the images in a team. The team is specified by the value of *image-team* and shall include the executing
8 image. All images of the team shall execute a SYNC TEAM statement with a value of *image-team* that
9 was constructed by corresponding invocations of the intrinsic function FORM_TEAM for the team. They
10 do not commence executing subsequent statements until all images in the team have executed a SYNC
11 TEAM statement for the team an equal number of times since FORM_TEAM was invoked for the team.
12 If images M and T are any two members of the team, the segments that execute before the statement
13 on image M precede the segments that execute after the statement on image T.

14 Execution of an OPEN statement with a TEAM= specifier, a CLOSE statement for a unit whose connect
15 team consists of more than one image, or a CALL statement for a collective subroutine is interpreted
16 as if an execution of a SYNC TEAM statement for the team occurred at the beginning and end of
17 execution of the statement. The team is identified by the value of *image-team* in the statement, is the
18 set of all images for a collective subroutine with no TEAM argument, or is the connect team for the
19 CLOSE statement.

NOTE 8.36

Execution of the FORM_TEAM intrinsic subroutine also performs a team synchronization.

J3 internal note**Unresolved Technical Issue 086**

What happens if *image-team* identifies a team that does not include the image executing the SYNC TEAM statement? (For example, it could be an empty image set.) It would be friendly for this to raise an error in the statement rather than making the program non-conformant.

NOTE 8.37

In this example the images are divided into two teams, one for an ocean calculation and one for an atmosphere calculation.

```

USE, INTRINSIC :: ISO_FORTRAN_ENV
TYPE(IMAGE_TEAM) :: TEAM
INTEGER :: N2, STEP, NSTEPS
LOGICAL :: OCEAN

N2 = NUM_IMAGES()/2
OCEAN = (THIS_IMAGE()<=N2)
IF (OCEAN) THEN
    CALL FORM_TEAM (TEAM, [ (I, I=1,N2) ] )
ELSE
    CALL FORM_TEAM (TEAM, [ (I, I=N2+1,NUM_IMAGES()) ] )
END IF
: ! Initial calculation
SYNC ALL
DO STEP = 1, NSTEPS
    IF (OCEAN) THEN
        DO
            : ! Ocean calculation
            SYNC TEAM (TEAM)
            IF ( ... ) EXIT ! Ready to swap data
        END DO
    ELSE
        DO
            : ! Atmosphere calculation
            SYNC TEAM (TEAM)
            IF ( ... ) EXIT ! Ready to swap data
        END DO
    END IF
    SYNC ALL
    : ! Swap data
END DO

```

In the inner loops, each set of images first works entirely with its own data and each image synchronizes with the rest of its team. The number of synchronizations for the ocean team might differ from the number for the atmosphere team. The SYNC ALL statement that follows is needed to ensure that both teams have done their calculations before data are swapped.

1 **8.5.4 SYNC IMAGES statement**

2 R861 *sync-images-stmt* **is** SYNC IMAGES (*image-set* [, *sync-stat-list*])

3 R862 *image-set* **is** *int-expr*
 4 **or** *

5 C850 An *image-set* that is an *int-expr* shall be scalar or of rank one.

6 If *image-set* is an array expression, the value of each element shall be positive and not greater than the
 7 number of images, and there shall be no repeated values.

8 If *image-set* is a scalar expression, its value shall be positive and not greater than the number of images.

- 1 An *image-set* that is an asterisk specifies all images.
- 2 Execution of a SYNC IMAGES statement performs a synchronization of the image with each of the
- 3 other images in the *image-set*. Execution on an image, M, of the segment following the SYNC IMAGES
- 4 statement is delayed until each other image T in the *image-set* has executed a SYNC IMAGES statement
- 5 with M in its *image-set* as many times as image M has executed a SYNC IMAGES statement with T in
- 6 the *image-set*. The segments that executed before the SYNC IMAGES statement on image M precede
- 7 the segments that execute after the SYNC IMAGES statement on image T.

NOTE 8.38

A SYNC IMAGES statement that specifies the single image value THIS_IMAGE() in its image set is allowed. This simplifies writing programs for an arbitrary number of images by allowing correct execution in the limiting case of the number of images being equal to one.

NOTE 8.39

Execution of SYNC IMAGES (*) on all images has the same effect as execution of SYNC ALL on all images, but SYNC ALL might have better performance. SYNC IMAGES statements are not required to specify the entire image set, or even the same image set, on all images participating in the synchronization.

In the following example, image 1 will wait for each of the other images to complete its use of the data. The other images wait for image 1 to set up the data, but do not wait on any of the other images.

```
IF (THIS_IMAGE() == 1) then
  ! Set up co-array data needed by all other images
  SYNC IMAGES(*)
ELSE
  SYNC IMAGES(1)
  ! Use the data set up by image 1
END IF
```

NOTE 8.40

Execution of a SYNC TEAM statement causes all the images of the team to wait for each other. There might, however, be situations where this is not efficient. 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

NOTE 8.40 (cont.)

finish and then the others will finish one by one.

The SYNC IMAGES syntax involves *image-set* rather than *image-team* to allow the set of images to vary from image to image.

1 8.5.5 NOTIFY and QUERY statements

2 R863 *notify-stmt* **is** NOTIFY (*image-set* [, *sync-stat-list*])

3 R864 *query-stmt* **is** QUERY (*image-set* [, *query-spec-list*])

4 R865 *query-spec* **is** READY = *scalar-logical-variable*
5 **or** *sync-stat*

6 C851 (R864) No specifier shall appear more than once in a given *query-spec-list*.

7 Execution on image M of a NOTIFY statement with a different image T in its *image-set* increments by
8 1 a record of the number of times, $N_{M \rightarrow T}$, image M executed such a NOTIFY statement.

9 A QUERY statement is **blocking** if and only if it has no READY= specifier. A QUERY statement is
10 **satisfied** on completion of its execution if and only if it is a blocking QUERY statement or it set the
11 variable specified by its READY= specifier to true.

12 Let $Q_{M \leftarrow T}$ denote the number of times image M has completed the execution of a satisfied QUERY
13 statement with a different image T in its image set. Completion of execution on image M of a blocking
14 QUERY statement is delayed until, for each different T in its image set, $N_{T \rightarrow M} > Q_{M \leftarrow T}$.

15 Execution of a non-blocking QUERY statement on image M causes the *scalar-logical-variable* of its
16 READY= specifier to be assigned the value false if, for a different image T in the image set, $N_{T \rightarrow M} \leq$
17 $Q_{M \leftarrow T}$; otherwise, true is assigned.

18 A NOTIFY statement execution on image T and a satisfied QUERY statement on image M correspond
19 if and only if

- 20 • the NOTIFY statement's image set includes image M,
- 21 • the QUERY statement's image set includes image T, and
- 22 • after execution of both statements has completed, $N_{T \rightarrow M} = Q_{M \leftarrow T}$.

23 Segments on an image executed before the execution of a NOTIFY statement precede the segments on
24 other images that follow its corresponding QUERY statements.

NOTE 8.41

The NOTIFY and QUERY statements can be used to order statement executions between a producer and consumer image.

```
INTEGER,PARAMETER :: PRODUCER = 1, CONSUMER = 2
INTEGER :: VALUE[*]
LOGICAL :: READY
```

```
SELECT CASE (THIS_IMAGE())
CASE (PRODUCER)
  VALUE[CONSUMER] = 3
```

NOTE 8.41 (cont.)

```

    NOTIFY (CONSUMER)
CASE (CONSUMER)
    WaitLoop: DO
        QUERY (PRODUCER,READY=READY)
        IF (READY) EXIT WaitLoop
        ! Statements neither referencing VALUE[CONSUMER], nor causing it to
        ! become defined or undefined
    END DO WaitLoop
    ! references to VALUE
CASE DEFAULT
    ! Statements neither referencing VALUE[CONSUMER], nor causing it to
    ! become defined or undefined
END SELECT

```

Unlike SYNC IMAGES statements, the number of notifications and corresponding queries may be unequal. A program can complete with an excess number of notifies.

NOTE 8.42

NOTIFY/QUERY pairs can be used in place of SYNC ALL and SYNC IMAGES to achieve better load balancing and allow one image to proceed with calculations while another image is catching up. For example,

```

IF (THIS_IMAGE()==1) THEN
    DO I=1,100
        ...      ! Primary processing of column I
        NOTIFY(2) ! Done with column I
    END DO
    SYNC IMAGES(2)
ELSE IF (THIS_IMAGE()==2) THEN
    DO I=1,100
        QUERY(1) ! Wait until image 1 is done with column I
        ...      ! Secondary processing of column I
    END DO
    SYNC IMAGES(1)
END IF

```

1 8.5.6 SYNC MEMORY statement

- 2 The SYNC MEMORY statement provides a means of dividing a segment on an image into two segments,
 3 each of which can be ordered in some user-defined way with respect to segments on other images.

4 R866 *sync-memory-stmt* is SYNC MEMORY [([*sync-stat-list*])]

NOTE 8.43

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.

- 1 All of the other image control statements include the effect of executing a SYNC MEMORY statement.
- 2 In addition, the other image control statements cause some form of cooperation with other images for
- 3 the purpose of ordering execution between images.

NOTE 8.44

A common example of user-written code that can be used in conjunction with SYNC MEMORY to implement specialized schemes for segment ordering is the spin-wait loop. For example:

```

LOGICAL, VOLATILE :: LOCKED[*] = .TRUE.
INTEGER :: IAM, P, Q

IAM = THIS_IMAGE()
IF (IAM == P) THEN
    ! Preceding segment
    SYNC MEMORY                ! A
    LOCKED[Q] = .FALSE.        ! segment  $P_i$ 
    SYNC MEMORY                ! B
ELSE IF (IAM == Q) THEN
    DO WHILE (LOCKED); END DO  ! segment  $Q_j$ 
    SYNC MEMORY                ! C
    ! Subsequent segment
END IF

```

Here, image Q does not complete the segment Q_j until image P executes segment P_i . This ensures that executions of segments before P_i on image P precede executions of segments on image Q after Q_j .

The first SYNC MEMORY statement (A) ensures that the compiler does not reorder the following statement (locking) with the previous statements, since the lock should be freed only after the work has been completed.

The definition of LOCKED[Q] might be deferred to the end of segment P_i . The second SYNC MEMORY statement (B) ends that segment immediately after the definition, minimizing any delay in releasing the lock in segment Q_j .

The third SYNC MEMORY statement (C) marks the beginning of a new segment, informing the compiler that the values of co-array variables referenced in that segment might have been changed by other images in preceding segments, so need to be loaded from memory.

NOTE 8.45

As a second example, the user might have access to an external procedure that performs synchronization between images. That library procedure will not necessarily 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

```

NOTE 8.45 (cont.)

```
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.

1 8.5.7 STAT= and ERRMSG= specifiers in image execution control statements

- 2 If the STAT= specifier appears, successful execution of the SYNC ALL, SYNC TEAM, SYNC IMAGES,
3 SYNC MEMORY, NOTIFY, or QUERY statement causes the specified variable to become defined with
4 the value zero. If an error occurs during execution of one of these statements, the variable becomes
5 defined with a processor-dependent positive integer value. If an error condition occurs during execution
6 of a SYNC ALL, SYNC TEAM, SYNC IMAGES, SYNC MEMORY, NOTIFY, or QUERY statement
7 that does not contain the STAT= specifier, execution of all images is terminated.
- 8 If the ERRMSG= specifier appears and an error condition occurs during execution of the SYNC ALL,
9 SYNC TEAM, SYNC IMAGES, SYNC MEMORY, NOTIFY, or QUERY statement, the processor shall
10 assign an explanatory message to the specified variable. If no such condition occurs, the processor shall
11 not change the value of the variable.

NOTE 8.46

Which errors, if any, are diagnosed is processor dependent. The processor might check that a valid set of images has been provided, with no out-of-range or repeated values. It might test for network time-outs. While the overall program would probably not be able to recover from the failure of an image, it could perhaps provide information on what failed and be able to save some of the program state to a file.

9 Input/output statements

Input statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage. This process is called **reading**. **Output statements** provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is called **writing**. Some input/output statements specify that editing of the data is to be performed.

In addition to the statements that transfer data, there are auxiliary input/output statements to manipulate the external medium, or to describe or inquire about the properties of the connection to the external medium.

The input/output statements are the OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, ENDFILE, REWIND, FLUSH, WAIT, and INQUIRE statements.

The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT statement are **data transfer output statements**. The OPEN statement and the CLOSE statement are **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The BACKSPACE, ENDFILE, and REWIND statements are **file positioning statements**.

A file is composed of either a sequence of file storage units (9.2.4) or a sequence of records, which provide an extra level of organization to the file. A file composed of records is called a **record file**. A file composed of file storage units is called a **stream file**. A processor may allow a file to be viewed both as a record file and as a stream file; in this case the relationship between the file storage units when viewed as a stream file and the records when viewed as a record file is processor dependent.

A file is either an external file (9.2) or an internal file (9.3).

9.1 Records

A **record** is a sequence of values or a sequence of characters. For example, a line on a terminal is usually considered to be a record. However, a record does not necessarily correspond to a physical entity. There are three kinds of records:

- (1) formatted;
- (2) unformatted;
- (3) endfile.

NOTE 9.1

What is called a “record” in Fortran is commonly called a “logical record”. There is no concept in Fortran of a “physical record.”

9.1.1 Formatted record

A **formatted record** consists of a sequence of characters that are representable in the processor; however, a processor may prohibit some control characters (3.1) 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 may be read or written only by formatted input/output statements.

1 Formatted records may be prepared by means other than Fortran.

2 9.1.2 Unformatted record

3 An **unformatted record** consists of a sequence of values in a processor-dependent form and may contain
4 data of any type or may contain no data. The length of an unformatted record is measured in file storage
5 units (9.2.4) and depends on the output list (9.5.3) used when it is written, as well as on the processor
6 and the external medium. The length may be zero. Unformatted records may be read or written only
7 by unformatted input/output statements.

8 9.1.3 Endfile record

9 An **endfile record** is written explicitly by the ENDFILE statement; the file shall be connected for
10 sequential access. An endfile record is written implicitly to a file connected for sequential access when
11 the most recent data transfer statement referring to the file is a data transfer output statement, no
12 intervening file positioning statement referring to the file has been executed, and

- 13 (1) a REWIND or BACKSPACE statement references the unit to which the file is connected,
14 or
- 15 (2) the unit is closed, either explicitly by a CLOSE statement, implicitly by a program termi-
16 nation not caused by an error condition, or implicitly by another OPEN statement for the
17 same unit.

18 An endfile record may occur only as the last record of a file. An endfile record does not have a length
19 property.

NOTE 9.2

An endfile record does not necessarily have any physical embodiment. The processor may use a record count or other means to register the position of the file at the time an ENDFILE statement is executed, so that it can take appropriate action when that position is reached again during a read operation. The endfile record, however it is implemented, is considered to exist for the BACKSPACE statement (9.7.2).

20 9.2 External files

21 An **external file** is any file that exists in a medium external to the program.

22 At any given time, there is a processor-dependent set of allowed **access methods**, a processor-dependent
23 set of allowed **forms**, a processor-dependent set of allowed **actions**, and a processor-dependent set of
24 allowed **record lengths** for a file.

NOTE 9.3

For example, the processor-dependent set of allowed actions for a printer would likely include the write action, but not the read action.

25 A file may have a name; a file that has a name is called a **named file**. The name of a named file is
26 represented by a character string value. The set of allowable names for a file is processor dependent.

NOTE 9.4

Named files that are opened with the TEAM= specifier (9.4.5.17) have the same name on each image of the team. Apart from this, whether a named file on one image is the same as a file with the same name on another image is processor dependent. For code portability, if different files are

NOTE 9.4 (cont.)

needed on each image, different file names should be used. One technique is to incorporate the image number as part of the name.

- 1 An external file that is connected to a unit has a **position** property (9.2.3).

NOTE 9.5

For more explanatory information on external files, see C.6.1.

2 **9.2.1 File existence**

- 3 At any given time, there is a processor-dependent set of external files that **exist** for a program. A file
4 may be known to the processor, yet not exist for a program at a particular time.

NOTE 9.6

Security reasons may prevent a file from existing for a program. A newly created file may exist but contain no records.

- 5 To create a file means to cause a file to exist that did not exist previously. To delete a file means to
6 terminate the existence of the file.

- 7 All input/output statements may refer to files that exist. An INQUIRE, OPEN, CLOSE, WRITE,
8 PRINT, REWIND, FLUSH, or ENDFILE statement also may refer to a file that does not exist. Execu-
9 tion of a WRITE, PRINT, or ENDFILE statement referring to a preconnected file that does not exist
10 creates the file.

11 **9.2.2 File access**

- 12 There are three methods of accessing the data of an external file: sequential, direct, and stream. Some
13 files may have more than one allowed access method; other files may be restricted to one access method.

NOTE 9.7

For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

- 14 The method of accessing a file is determined when the file is connected to a unit (9.4.3) or when the file
15 is created if the file is preconnected (9.4.4).

16 **9.2.2.1 Sequential access**

- 17 **Sequential access** is a method of accessing the records of an external record file in order.

- 18 When connected for sequential access, an external file has the following properties.

- 19 (1) The order of the records is the order in which they were written if the direct access method
20 is not a member of the set of allowed access methods for the file. If the direct access method
21 is also a member of the set of allowed access methods for the file, the order of the records
22 is the same as that specified for direct access. In this case, the first record accessible by
23 sequential access is the record whose record number is 1 for direct access. The second record
24 accessible by sequential access is the record whose record number is 2 for direct access, etc.
25 A record that has not been written since the file was created shall not be read.
- 26 (2) The records of the file are either all formatted or all unformatted, except that the last record
27 of the file may be an endfile record. Unless the previous reference to the file was a data

- 1 transfer output statement, the last record, if any, of the file shall be an endfile record.
- 2 (3) The records of the file shall be read or written only by sequential access input/output
- 3 statements.
- 4 (4) Each record shall be read or written by a single image. The processor shall ensure that once
- 5 an image commences transferring the data of a record to the file, no other image transfers
- 6 data to the file until the whole record has been transferred.

7 9.2.2.2 Direct access

8 **Direct access** is a method of accessing the records of an external record file in arbitrary order.

9 When connected for direct access, an external file has the following properties.

- 10 (1) Each record of the file is uniquely identified by a positive integer called the **record number**.
- 11 The record number of a record is specified when the record is written. Once established,
- 12 the record number of a record can never be changed. The order of the records is the order
- 13 of their record numbers.

NOTE 9.8

A record cannot be deleted; however, a record may be rewritten.

- 14 (2) The records of the file are either all formatted or all unformatted. If the sequential access
- 15 method is also a member of the set of allowed access methods for the file, its endfile record,
- 16 if any, is not considered to be part of the file while it is connected for direct access. If the
- 17 sequential access method is not a member of the set of allowed access methods for the file,
- 18 the file shall not contain an endfile record.
- 19 (3) The records of the file shall be read or written only by direct access input/output statements.
- 20 (4) All records of the file have the same length.
- 21 (5) Records need not be read or written in the order of their record numbers. Any record may
- 22 be written into the file while it is connected to a unit. For example, it is permissible to write
- 23 record 3, even though records 1 and 2 have not been written. Any record may be read from
- 24 the file while it is connected to a unit, provided that the record has been written since the
- 25 file was created, and if a READ statement for this connection is permitted.
- 26 (6) The records of the file shall not be read or written using list-directed formatting (10.10),
- 27 namelist formatting (10.11), or a nonadvancing input/output statement (9.2.3.1).

28 9.2.2.3 Stream access

29 **Stream access** is a method of accessing the file storage units (9.2.4) of an external stream file.

30 The properties of an external file connected for stream access depend on whether the connection is for

31 unformatted or formatted access.

32 When connected for unformatted stream access, an external file has the following properties.

- 33 (1) The file storage units of the file shall be read or written only by stream access input/output
- 34 statements.
- 35 (2) Each file storage unit in the file is uniquely identified by a positive integer called the position.
- 36 The first file storage unit in the file is at position 1. The position of each subsequent file
- 37 storage unit is one greater than that of its preceding file storage unit.
- 38 (3) If it is possible to position the file, the file storage units need not be read or written in
- 39 order of their position. For example, it might be permissible to write the file storage unit
- 40 at position 3, even though the file storage units at positions 1 and 2 have not been written.
- 41 Any file storage unit may be read from the file while it is connected to a unit, provided that

1 the file storage unit has been written since the file was created, and if a READ statement
2 for this connection is permitted.

3 When connected for formatted stream access, an external file has the following properties.

- 4 (1) Some file storage units of the file may contain record markers; this imposes a record structure
5 on the file in addition to its stream structure. There might or might not be a record marker
6 at the end of the file. If there is no record marker at the end of the file, the final record is
7 incomplete.
- 8 (2) No maximum length (9.4.5.13) is applicable to these records.
- 9 (3) Writing an empty record with no record marker has no effect.

NOTE 9.9

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.

- 10 (4) The file storage units of the file shall be read or written only by formatted stream access
11 input/output statements.
- 12 (5) Each file storage unit in the file is uniquely identified by a positive integer called the position.
13 The first file storage unit in the file is at position 1. The relationship between positions of
14 successive file storage units is processor dependent; not all positive integers need correspond
15 to valid positions.
- 16 (6) If it is possible to position the file, the file position can be set to a position that was
17 previously identified by the POS= specifier in an INQUIRE statement.

NOTE 9.10

There may be some character positions in the file that do not correspond to characters written; this is because on some processors a record marker may be written to the file as a carriage-return/line-feed or other sequence. The means of determining the position in a file connected for stream access is via the POS= specifier in an INQUIRE statement (9.9.2.21).

- 18 (7) A processor may prohibit some control characters (3.1) from appearing in a formatted stream
19 file.

20 9.2.3 File position

21 Execution of certain input/output statements affects the position of an external file. Certain circum-
22 stances can cause the position of a file to become indeterminate.

23 The **initial point** of a file is the position just before the first record or file storage unit. The **terminal**
24 **point** is the position just after the last record or file storage unit. If there are no records or file storage
25 units in the file, the initial point and the terminal point are the same position.

26 If a record file is positioned within a record, that record is the **current record**; otherwise, there is no
27 current record.

28 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
29 between the $(i - 1)$ th record and the i th record, the $(i - 1)$ th record is the **preceding record**. If $n \geq 1$
30 and the file is positioned at its terminal point, the preceding record is the n th and last record. If $n = 0$
31 or if a file is positioned at its initial point or within the first record, there is no preceding record.

32 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
33 $(i + 1)$ th record is the **next record**. If $n \geq 1$ and the file is positioned at its initial point, the first record
34 is the next record. If $n = 0$ or if a file is positioned at its terminal point or within the n th (last) record,
35 there is no next record.

1 For a file connected for stream access, the file position is either between two file storage units, at the
2 initial point of the file, at the terminal point of the file, or undefined.

3 **9.2.3.1 Advancing and nonadvancing input/output**

4 An **advancing input/output statement** always positions a record file after the last record read or
5 written, unless there is an error condition.

6 A **nonadvancing input/output statement** may position a record file at a character position within
7 the current record, or a subsequent record (10.8.2). Using nonadvancing input/output, it is possible to
8 read or write a record of the file by a sequence of input/output statements, each accessing a portion
9 of the record. It is also possible to read variable-length records and be notified of their lengths. If a
10 nonadvancing output statement leaves a file positioned within a current record and no further output
11 statement is executed for the file before it is closed or a BACKSPACE, ENDFILE, or REWIND statement
12 is executed for it, the effect is as if the output statement were the corresponding advancing output
13 statement.

14 **9.2.3.2 File position prior to data transfer**

15 The positioning of the file prior to data transfer depends on the method of access: sequential, direct, or
16 stream.

17 For sequential access on input, if there is a current record, the file position is not changed. Otherwise,
18 the file is positioned at the beginning of the next record and this record becomes the current record.
19 Input shall not occur if there is no next record or if there is a current record and the last data transfer
20 statement accessing the file performed output.

21 If the file contains an endfile record, the file shall not be positioned after the endfile record prior to data
22 transfer. However, a REWIND or BACKSPACE statement may be used to reposition the file.

23 For sequential access on output, if there is a current record, the file position is not changed and the
24 current record becomes the last record of the file. Otherwise, a new record is created as the next record
25 of the file; this new record becomes the last and current record of the file and the file is positioned at
26 the beginning of this record.

27 For direct access, the file is positioned at the beginning of the record specified by the REC= specifier.
28 This record becomes the current record.

29 For stream access, the file is positioned immediately before the file storage unit specified by the POS=
30 specifier; if there is no POS= specifier, the file position is not changed.

31 File positioning for child data transfer statements is described in 9.5.4.7.

32 **9.2.3.3 File position after data transfer**

33 If an error condition (9.10) occurred, the position of the file is indeterminate. If no error condition
34 occurred, but an end-of-file condition (9.10) occurred as a result of reading an endfile record, the file is
35 positioned after the endfile record.

36 For unformatted stream access, if no error condition occurred, the file position is not changed. For
37 unformatted stream output, if the file position exceeds the previous terminal point of the file, the
38 terminal point is set to the file position.

NOTE 9.11

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.

- 1 For formatted stream input, if an end-of-file condition occurred, the file position is not changed.
- 2 For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-record
3 condition (9.10) occurred, the file is positioned after the record just read. If no error condition, end-of-
4 file condition, or end-of-record condition occurred in a nonadvancing input statement, the file position
5 is not changed. If no error condition occurred in a nonadvancing output statement, the file position is
6 not changed.
- 7 In all other cases, the file is positioned after the record just read or written and that record becomes the
8 preceding record.
- 9 For a formatted stream output statement, if no error condition occurred, the terminal point of the file
10 is set to the highest-numbered position to which data was transferred by the statement.

NOTE 9.12

The highest-numbered position might not be the current one if the output involved T or TL edit descriptors (10.8.1.1) and the statement is a nonadvancing output statement.

11 9.2.4 File storage units

- 12 A **file storage unit** is the basic unit of storage in a stream file or an unformatted record file. It is the
13 unit of file position for stream access, the unit of record length for unformatted files, and the unit of file
14 size for all external files.
- 15 Every value in a stream file or an unformatted record file shall occupy an integer number of file storage
16 units; if the stream or record file is unformatted, this number shall be the same for all scalar values of
17 the same type and type parameters. The number of file storage units required for an item of a given type
18 and type parameters may be determined using the IOLENGTH= specifier of the INQUIRE statement
19 (9.9.3).
- 20 For a file connected for unformatted stream access, the processor shall not have alignment restrictions
21 that prevent a value of any type from being stored at any positive integer file position.
- 22 The number of bits in a file storage unit is given by the constant FILE_STORAGE_SIZE (13.8.2.6)
23 defined in the intrinsic module ISO_FORTRAN_ENV. It is recommended that the file storage unit be
24 an 8-bit octet where this choice is practical.

NOTE 9.13

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.

25 9.3 Internal files

1 Internal files provide a means of transferring and converting data from internal storage to internal
2 storage.

3 An internal file is a record file with the following properties.

- 4 (1) The file is a variable of default, ASCII, or ISO 10646 character type that is not an array
5 section with a vector subscript.
- 6 (2) A record of an internal file is a scalar character variable.
- 7 (3) If the file is a scalar character variable, it consists of a single record whose length is the same
8 as the length of the scalar character variable. If the file is a character array, it is treated
9 as a sequence of character array elements. Each array element, if any, is a record of the
10 file. The ordering of the records of the file is the same as the ordering of the array elements
11 in the array (6.2.2.1) or the array section (6.2.2.2). Every record of the file has the same
12 length, which is the length of an array element in the array.
- 13 (4) A record of the internal file becomes defined by writing the record. If the number of
14 characters written in a record is less than the length of the record, the remaining portion
15 of the record is filled with blanks. The number of characters to be written shall not exceed
16 the length of the record.
- 17 (5) A record may be read only if the record is defined.
- 18 (6) A record of an internal file may become defined (or undefined) by means other than an
19 output statement. For example, the character variable may become defined by a character
20 assignment statement.
- 21 (7) An internal file is always positioned at the beginning of the first record prior to data transfer,
22 except for child data transfer statements (9.5.4.7). This record becomes the current record.
- 23 (8) The initial value of a connection mode (9.4.1) is the value that would be implied by an
24 initial OPEN statement without the corresponding keyword.
- 25 (9) Reading and writing records shall be accomplished only by sequential access formatted
26 input/output statements.
- 27 (10) An internal file shall not be specified as the unit in a file connection statement or a file
28 positioning statement.

29 9.4 File connection

30 A **unit**, specified by an *io-unit*, provides a means for referring to a file.

31 R901 *io-unit* **is** *file-unit-number*
32 **or** *
33 **or** *internal-file-variable*

34 R902 *file-unit-number* **is** *scalar-int-expr*

35 R903 *internal-file-variable* **is** *char-variable*

36 C901 (R903) The *char-variable* shall not be an array section with a vector subscript.

37 C902 (R903) The *char-variable* shall be of type default character, ASCII character, or ISO 10646
38 character.

39 A unit is either an external unit or an internal unit. An **external unit** is used to refer to an external file
40 and is specified by an asterisk or a *file-unit-number*. The value of *file-unit-number* shall be nonnegative,
41 equal to one of the named constants INPUT_UNIT, OUTPUT_UNIT, or ERROR_UNIT of the ISO_
42 FORTRAN_ENV module (13.8.2), or a NEWUNIT value (9.4.5.10). An **internal unit** is used to refer
43 to an internal file and is specified by an *internal-file-variable* or a *file-unit-number* whose value is equal
44 to the **unit** argument of an active derived-type input/output procedure (9.5.4.7). The value of a *file-*

- 1 *unit-number* shall identify a valid unit.
- 2 The external unit identified by a particular value of a *scalar-int-expr* is the same external unit in all
- 3 program units of the program, and on all images.

NOTE 9.14

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 An asterisk identifies particular processor-dependent external units that are preconnected for format-
- 5 ted sequential access (9.5.4.2). These units are also identified by unit numbers defined by the named
- 6 constants `INPUT_UNIT` and `OUTPUT_UNIT` of the `ISO_FORTRAN_ENV` module (13.8.2).
- 7 This part of ISO/IEC 1539 identifies a processor-dependent external unit for the purpose of error re-
- 8 porting. This unit shall be preconnected for sequential formatted output. The processor may define this
- 9 to be the same as the output unit identified by an asterisk. This unit is also identified by a unit number
- 10 defined by the named constant `ERROR_UNIT` of the `ISO_FORTRAN_ENV` intrinsic module.

9.4.1 Connection modes

- 12 A connection for formatted input/output has several changeable modes: the blank interpretation mode
- 13 (10.8.6), delimiter mode (10.10.4, 10.11.4.1), sign mode (10.8.4), decimal edit mode (10.8.8), I/O round-
- 14 ing mode (10.7.2.3.7), pad mode (9.5.4.4.2), and scale factor (10.8.5). A connection for unformatted
- 15 input/output has no changeable modes.

- 16 Values for the modes of a connection are established when the connection is initiated. If the connection
- 17 is initiated by an `OPEN` statement, the values are as specified, either explicitly or implicitly, by the
- 18 `OPEN` statement. If the connection is initiated other than by an `OPEN` statement (that is, if the file is
- 19 an internal file or preconnected file) the values established are those that would be implied by an initial
- 20 `OPEN` statement without the corresponding keywords.

- 21 The scale factor cannot be explicitly specified in an `OPEN` statement; it is implicitly 0.

- 22 The modes of a connection to an external file may be changed by a subsequent `OPEN` statement that
- 23 modifies the connection.

- 24 The modes of a connection may be temporarily changed by a corresponding keyword specifier in a
- 25 data transfer statement or by an edit descriptor. Keyword specifiers take effect at the beginning of
- 26 execution of the data transfer statement. Edit descriptors take effect when they are encountered in
- 27 format processing. When a data transfer statement terminates, the values for the modes are reset to the
- 28 values in effect immediately before the data transfer statement was executed.

9.4.2 Unit existence

- 30 At any given time, there is a processor-dependent set of external units that **exist** for a program.

- 31 All input/output statements may refer to units that exist. The `CLOSE`, `INQUIRE`, and `WAIT` state-

1 ments also may refer to units that do not exist.

2 **9.4.3 Connection of a file to a unit**

3 An external unit has a property of being **connected** or not connected. If connected, it refers to an
4 external file. An external unit may become connected by preconnection or by the execution of an OPEN
5 statement. The property of connection is symmetric; the unit is connected to a file if and only if the file
6 is connected to the unit.

7 Every input/output statement except an OPEN, CLOSE, INQUIRE, or WAIT statement shall refer to
8 a unit that is connected to a file and thereby make use of or affect that file.

9 A file may be connected and not exist (9.2.1).

NOTE 9.15

An example is a preconnected external file that has not yet been written.

10 A unit shall not be connected to more than one file at the same time, and a file shall not be connected to
11 more than one unit at the same time. However, means are provided to change the status of an external
12 unit and to connect a unit to a different file.

13 This part of ISO/IEC 1539 defines means of portable interoperability with C. C streams are described
14 in 7.19.2 of the C International Standard. Whether a unit can be connected to a file that is also
15 connected to a C stream is processor dependent. If a unit is connected to a file that is also connected to
16 a C stream, the results of performing input/output operations on such a file are processor dependent.
17 It is processor dependent whether the files connected to the units INPUT_UNIT, OUTPUT_UNIT,
18 and ERROR_UNIT correspond to the predefined C text streams standard input, standard output, and
19 standard error. If a procedure defined by means of Fortran and a procedure defined by means other than
20 Fortran perform input/output operations on the same external file, the results are processor dependent.
21 A procedure defined by means of Fortran and a procedure defined by means other than Fortran can
22 perform input/output operations on different external files without interference.

23 After an external unit has been disconnected by the execution of a CLOSE statement, it may be con-
24 nected again within the same program to the same file or to a different file. After an external file has
25 been disconnected by the execution of a CLOSE statement, it may be connected again within the same
26 program to the same unit or to a different unit.

NOTE 9.16

The only means of referencing a file that has been disconnected is by the appearance of its name
in an OPEN or INQUIRE statement. There might be no means of reconnecting an unnamed file
once it is disconnected.

27 An internal unit is always connected to the internal file designated by the variable that identifies the
28 unit.

NOTE 9.17

For more explanatory information on file connection properties, see C.6.5.

29 **9.4.4 Preconnection**

30 **Preconnection** means that the unit is connected to a file at the beginning of execution of the program
31 and therefore it may be specified in input/output statements without the prior execution of an OPEN
32 statement.

1 9.4.5 OPEN statement

2 An **OPEN statement** initiates or modifies the connection between an external file and a specified unit.
 3 The OPEN statement may be used to connect an existing file to a unit, create a file that is preconnected,
 4 create a file and connect it to a unit, or change certain modes of a connection between a file and a unit.

5 An external unit may be connected by an OPEN statement in any program unit of a program and, once
 6 connected, a reference to it may appear in any program unit of the program.

7 If the file to be connected to the unit does not exist but is the same as the file to which the unit is
 8 preconnected, the modes specified by an OPEN statement become a part of the connection.

9 If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect
 10 is as if a CLOSE statement without a STATUS= specifier had been executed for the unit immediately
 11 prior to the execution of an OPEN statement.

12 If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted.
 13 If the FILE= specifier is not included in such an OPEN statement, the file to be connected to the unit
 14 is the same as the file to which the unit is already connected.

15 If the file to be connected to the unit is the same as the file to which the unit is connected, only the
 16 specifiers for changeable modes (9.4.1) may have values different from those currently in effect. If the
 17 POSITION= specifier appears in such an OPEN statement, the value specified shall not disagree with
 18 the current position of the file. If the STATUS= specifier is included in such an OPEN statement, it shall
 19 be specified with the value OLD. Execution of such an OPEN statement causes any new values of the
 20 specifiers for changeable modes to be in effect, but does not cause any change in any of the unspecified
 21 specifiers and the position of the file is unaffected. The ERR=, IOSTAT=, and IOMSG= specifiers from
 22 any previously executed OPEN statement have no effect on any currently executed OPEN statement.

23 A STATUS= specifier with a value of OLD is always allowed when the file to be connected to the unit is
 24 the same as the file to which the unit is connected. In this case, if the status of the file was SCRATCH
 25 before execution of the OPEN statement, the file will still be deleted when the unit is closed, and the
 26 file is still considered to have a status of SCRATCH.

27 If a file is already connected to a unit, an OPEN statement on that file with a different unit shall not
 28 be executed.

29	R904	<i>open-stmt</i>	is	OPEN (<i>connect-spec-list</i>)
30	R905	<i>connect-spec</i>	is	[UNIT =] <i>file-unit-number</i>
31			or	ACCESS = <i>scalar-default-char-expr</i>
32			or	ACTION = <i>scalar-default-char-expr</i>
33			or	ASYNCHRONOUS = <i>scalar-default-char-expr</i>
34			or	BLANK = <i>scalar-default-char-expr</i>
35			or	DECIMAL = <i>scalar-default-char-expr</i>
36			or	DELIM = <i>scalar-default-char-expr</i>
37			or	ENCODING = <i>scalar-default-char-expr</i>
38			or	ERR = <i>label</i>
39			or	FILE = <i>file-name-expr</i>
40			or	FORM = <i>scalar-default-char-expr</i>
41			or	IOMSG = <i>iomsg-variable</i>
42			or	IOSTAT = <i>scalar-int-variable</i>
43			or	NEWUNIT = <i>scalar-int-variable</i>
44			or	PAD = <i>scalar-default-char-expr</i>
45			or	POSITION = <i>scalar-default-char-expr</i>
46			or	RECL = <i>scalar-int-expr</i>

1 **or** ROUND = *scalar-default-char-expr*
2 **or** SIGN = *scalar-default-char-expr*
3 **or** STATUS = *scalar-default-char-expr*
4 **or** TEAM = *image-team*

5 R906 *file-name-expr* **is** *scalar-default-char-expr*

6 R907 *iomsg-variable* **is** *scalar-default-char-variable*

7 C903 No specifier shall appear more than once in a given *connect-spec-list*.

8 C904 (R904) If the NEWUNIT= specifier does not appear, a *file-unit-number* shall be specified; if
9 the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the
10 *connect-spec-list*.

11 C905 (R904) The *label* used in the ERR= specifier shall be the statement label of a branch target
12 statement that appears in the same scoping unit as the OPEN statement.

13 C906 (R904) If a NEWUNIT= specifier appears, a *file-unit-number* shall not appear.

14 If the STATUS= specifier has the value NEW or REPLACE, the FILE= specifier shall appear. If the
15 STATUS= specifier has the value SCRATCH, the FILE= specifier shall not appear. If the STATUS=
16 specifier has the value OLD, the FILE= specifier shall appear unless the unit is connected and the file
17 connected to the unit exists.

18 If the NEWUNIT= specifier appears in an OPEN statement, either the FILE= specifier shall appear,
19 or the STATUS= specifier shall appear with a value of SCRATCH. The unit identified by a NEWUNIT
20 value shall not be preconnected.

21 A specifier that requires a *scalar-default-char-expr* may have a limited list of character values. These
22 values are listed for each such specifier. Any trailing blanks are ignored. The value specified is without
23 regard to case. Some specifiers have a default value if the specifier is omitted.

24 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

NOTE 9.18

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
```

NOTE 9.19

For more explanatory information on the OPEN statement, see C.6.4.

25 **9.4.5.1 ACCESS= specifier in the OPEN statement**

26 The *scalar-default-char-expr* shall evaluate to SEQUENTIAL, DIRECT, or STREAM. The ACCESS=
27 specifier specifies the access method for the connection of the file as being sequential, direct, or stream.
28 If this specifier is omitted, the default value is SEQUENTIAL. For an existing file, the specified access
29 method shall be included in the set of allowed access methods for the file. For a new file, the processor
30 creates the file with a set of allowed access methods that includes the specified method.

31 **9.4.5.2 ACTION= specifier in the OPEN statement**

32 The *scalar-default-char-expr* shall evaluate to READ, WRITE, or READWRITE. READ specifies that
33 the WRITE, PRINT, and ENDFILE statements shall not refer to this connection. WRITE specifies

1 that READ statements shall not refer to this connection. READWRITE permits any input/output
2 statements to refer to this connection. If this specifier is omitted, the default value is processor dependent.
3 If READWRITE is included in the set of allowable actions for a file, both READ and WRITE also shall
4 be included in the set of allowed actions for that file. For an existing file, the specified action shall be
5 included in the set of allowed actions for the file. For a new file, the processor creates the file with a set
6 of allowed actions that includes the specified action.

7 **9.4.5.3 ASYNCHRONOUS= specifier in the OPEN statement**

8 The *scalar-default-char-expr* shall evaluate to YES or NO. If YES is specified, asynchronous input/output
9 on the unit is allowed. If NO is specified, asynchronous input/output on the unit is not allowed. If this
10 specifier is omitted, the default value is NO.

11 **9.4.5.4 BLANK= specifier in the OPEN statement**

12 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier is permitted only
13 for a connection for formatted input/output. It specifies the current value of the blank interpretation
14 mode (10.8.6, 9.5.2.6) for input for this connection. This mode has no effect on output. It is a changeable
15 mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the default
16 value is NULL.

17 **9.4.5.5 DECIMAL= specifier in the OPEN statement**

18 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier is per-
19 mitted only for a connection for formatted input/output. It specifies the current value of the decimal
20 edit mode (10.6, 10.8.8, 9.5.2.7) for this connection. This is a changeable mode (9.4.1). If this specifier
21 is omitted in an OPEN statement that initiates a connection, the default value is POINT.

22 **9.4.5.6 DELIM= specifier in the OPEN statement**

23 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= spec-
24 ifier is permitted only for a connection for formatted input/output. It specifies the current value of the
25 delimiter mode (9.5.2.8) for list-directed (10.10.4) and namelist (10.11.4.1) output for the connection.
26 This mode has no effect on input. It is a changeable mode (9.4.1). If this specifier is omitted in an
27 OPEN statement that initiates a connection, the default value is NONE.

28 **9.4.5.7 ENCODING= specifier in the OPEN statement**

29 The *scalar-default-char-expr* shall evaluate to UTF-8 or DEFAULT. The ENCODING= specifier is
30 permitted only for a connection for formatted input/output. The value UTF-8 specifies that the encoding
31 form of the file is UTF-8 as specified by ISO/IEC 10646-1:2000. Such a file is called a **Unicode** file,
32 and all characters therein are of ISO 10646 character type. The value UTF-8 shall not be specified if
33 the processor does not support the ISO 10646 character type. The value DEFAULT specifies that the
34 encoding form of the file is processor-dependent. If this specifier is omitted in an OPEN statement that
35 initiates a connection, the default value is DEFAULT.

36 **9.4.5.8 FILE= specifier in the OPEN statement**

37 The value of the FILE= specifier is the name of the file to be connected to the specified unit. Any trailing
38 blanks are ignored. The *file-name-expr* shall be a name that is allowed by the processor. If this specifier
39 is omitted and the unit is not connected to a file, the STATUS= specifier shall be specified with a value
40 of SCRATCH; in this case, the connection is made to a processor-dependent file. The interpretation of
41 case is processor dependent.

1 9.4.5.9 FORM= specifier in the OPEN statement

2 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
3 specifier is omitted, the default value is UNFORMATTED if the file is being connected for direct access
4 or stream access, and the default value is FORMATTED if the file is being connected for sequential
5 access. For an existing file, the specified form shall be included in the set of allowed forms for the file.
6 For a new file, the processor creates the file with a set of allowed forms that includes the specified form.
7

8 9.4.5.10 NEWUNIT= specifier in the OPEN statement

9 The variable is defined with a processor determined NEWUNIT value if no error occurs during the
10 execution of the OPEN statement. If an error occurs, the processor shall not change the value of the
11 variable.

12 A NEWUNIT value is a negative number, and shall not be equal to -1, any of the named constants
13 ERROR_UNIT, INPUT_UNIT, or OUTPUT_UNIT from the ISO_FORTRAN_ENV intrinsic module
14 (13.8.2), any value used by the processor for the unit argument to a user-defined derived-type in-
15 put/output procedure, nor any previous NEWUNIT value that identifies a file that is currently con-
16 nected.

17 9.4.5.11 PAD= specifier in the OPEN statement

18 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier is permitted only for a
19 connection for formatted input/output. It specifies the current value of the pad mode (9.5.4.4.2, 9.5.2.10)
20 for input for this connection. This mode has no effect on output. It is a changeable mode (9.4.1). If this
21 specifier is omitted in an OPEN statement that initiates a connection, the default value is YES.

22 9.4.5.12 POSITION= specifier in the OPEN statement

23 The *scalar-default-char-expr* shall evaluate to ASIS, REWIND, or APPEND. The connection shall be
24 for sequential or stream access. A new file is positioned at its initial point. REWIND positions an
25 existing file at its initial point. APPEND positions an existing file such that the endfile record is the
26 next record, if it has one. If an existing file does not have an endfile record, APPEND positions the
27 file at its terminal point. ASIS leaves the position unchanged if the file exists and already is connected.
28 ASIS leaves the position unspecified if the file exists but is not connected. If this specifier is omitted,
29 the default value is ASIS.

30 9.4.5.13 RECL= specifier in the OPEN statement

31 The value of the RECL= specifier shall be positive. It specifies the length of each record in a file being
32 connected for direct access, or specifies the maximum length of a record in a file being connected for
33 sequential access. This specifier shall not appear when a file is being connected for stream access. This
34 specifier shall appear when a file is being connected for direct access. If this specifier is omitted when
35 a file is being connected for sequential access, the default value is processor dependent. If the file is
36 being connected for formatted input/output, the length is the number of characters for all records that
37 contain only characters of type default character. When a record contains any nondefault characters,
38 the effect of the RECL= specifier is processor dependent. If the file is being connected for unformatted
39 input/output, the length is measured in file storage units. For an existing file, the value of the RECL=
40 specifier shall be included in the set of allowed record lengths for the file. For a new file, the processor
41 creates the file with a set of allowed record lengths that includes the specified value.

42 9.4.5.14 ROUND= specifier in the OPEN statement

1 The *scalar-default-char-expr* shall evaluate to one of UP, DOWN, ZERO, NEAREST, COMPATIBLE,
 2 or PROCESSOR_DEFINED. The ROUND= specifier is permitted only for a connection for formatted
 3 input/output. It specifies the current value of the I/O rounding mode (10.7.2.3.7, 9.5.2.13) for this
 4 connection. This is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that
 5 initiates a connection, the I/O rounding mode is processor dependent; it shall be one of the above modes.

NOTE 9.20

A processor is free to select any I/O rounding mode for the default mode. The mode might correspond to UP, DOWN, ZERO, NEAREST, or COMPATIBLE; or it might be a completely different I/O rounding mode.

6 9.4.5.15 SIGN= specifier in the OPEN statement

7 The *scalar-default-char-expr* shall evaluate to one of PLUS, SUPPRESS, or PROCESSOR_DEFINED.
 8 The SIGN= specifier is permitted only for a connection for formatted input/output. It specifies the
 9 current value of the sign mode (10.8.4, 9.5.2.14) for this connection. This is a changeable mode (9.4.1).
 10 If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PRO-
 11 CESSOR_DEFINED.

12 9.4.5.16 STATUS= specifier in the OPEN statement

13 The *scalar-default-char-expr* shall evaluate to OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. If
 14 OLD is specified, the file shall exist. If NEW is specified, the file shall not exist.

15 Successful execution of an OPEN statement with NEW specified creates the file and changes the status
 16 to OLD. If REPLACE is specified and the file does not already exist, the file is created and the status is
 17 changed to OLD. If REPLACE is specified and the file does exist, the file is deleted, a new file is created
 18 with the same name, and the status is changed to OLD. If SCRATCH is specified, the file is created
 19 and connected to the specified unit for use by the program but is deleted at the execution of a CLOSE
 20 statement referring to the same unit or at the normal termination of the program.

NOTE 9.21

SCRATCH shall not be specified with a named file.

21 If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default
 22 value is UNKNOWN.

23 9.4.5.17 TEAM= specifier in the OPEN statement

24 The *image-team* specifies the **connect team** for the unit, which is the set of images that are permitted
 25 to reference the unit. The team shall include the executing image. If there is no TEAM= specifier, the
 26 connect team consists of only the executing image.

27 All images in the connect team, and no others, shall execute the same OPEN statement with identical
 28 values for the *connect-specs*. There is an implicit team synchronization.

29 If the OPEN statement has a STATUS= specifier with the value SCRATCH, the processor shall connect
 30 the same file to the unit on all images in the connect team.

31 If the connect team contains more than one image, the OPEN statement shall

- 32 • specify direct access or
- 33 • specify sequential access and have an ACTION= specifier that evaluates to WRITE.

NOTE 9.22

Writing to a sequential file from more than one image without using synchronization is permitted, but is only useful for situations in which the ordering of records is unimportant. An example is for diagnostic output that is labeled with the image index.

- 1 Units connected to a preconnected file, other than the unit identified by the value `INPUT_UNIT` in
 2 the intrinsic module `ISO_FORTTRAN_ENV` have a connect team consisting of all the images. The unit
 3 identified by the value `INPUT_UNIT` in the intrinsic module `ISO_FORTTRAN_ENV` is preconnected with
 4 a connect team consisting of image 1 on image 1 and is not preconnected on other images.

NOTE 9.23

The input unit identified by `*` is therefore only available on the image with index one. File positioning statements and `READ` are not allowed for preconnected units, including units preconnected to `OUTPUT_UNIT` and `ERROR_UNIT`, if the number of images is greater than one. The `FLUSH` and `INQUIRE` statements are always allowed for preconnected units.

9.4.6 CLOSE statement

- 6 The **CLOSE statement** is used to terminate the connection of a specified unit to an external file.
- 7 Execution of a `CLOSE` statement for a unit may occur in any program unit of a program and need not
 8 occur in the same program unit as the execution of an `OPEN` statement referring to that unit.
- 9 Execution of a `CLOSE` statement performs a wait operation for any pending asynchronous data transfer
 10 operations for the specified unit.
- 11 Execution of a `CLOSE` statement specifying a unit that does not exist or has no file connected to it is
 12 permitted and affects no file or unit.
- 13 After a unit has been disconnected by execution of a `CLOSE` statement, it may be connected again
 14 within the same program, either to the same file or to a different file. After a named file has been
 15 disconnected by execution of a `CLOSE` statement, it may be connected again within the same program,
 16 either to the same unit or to a different unit, provided that the file still exists.
- 17 At normal termination of execution of a program, all units that are connected are closed. Each unit
 18 is closed with status `KEEP` unless the file status prior to termination of execution was `SCRATCH`, in
 19 which case the unit is closed with status `DELETE`.

NOTE 9.24

The effect is as though a `CLOSE` statement without a `STATUS=` specifier were executed on each connected unit.

20	R908 <i>close-stmt</i>	is	<code>CLOSE (<i>close-spec-list</i>)</code>
21	R909 <i>close-spec</i>	is	<code>[UNIT =] <i>file-unit-number</i></code>
22		or	<code>IOSTAT = <i>scalar-int-variable</i></code>
23		or	<code>IOMSG = <i>iomsg-variable</i></code>
24		or	<code>ERR = <i>label</i></code>
25		or	<code>STATUS = <i>scalar-default-char-expr</i></code>
- 26 C907 No specifier shall appear more than once in a given *close-spec-list*.
- 27 C908 A *file-unit-number* shall be specified in a *close-spec-list*; if the optional characters `UNIT=` are
 28 omitted, the *file-unit-number* shall be the first item in the *close-spec-list*.

- 1 C909 (R909) The *label* used in the ERR= specifier shall be the statement label of a branch target
 2 statement that appears in the same scoping unit as the CLOSE statement.
- 3 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 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.
- 6 If an image executes a CLOSE statement, all images in the connect team of the unit specified shall execute
 7 a CLOSE statement for the unit with the same disposition. There is an implicit team synchronization
 8 associated with the execution of a CLOSE statement for a unit with a connect team that has more than
 9 one image (8.5.3).

NOTE 9.25

An example of a CLOSE statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

NOTE 9.26

For more explanatory information on the CLOSE statement, see C.6.6.

10 9.4.6.1 STATUS= specifier in the CLOSE statement

- 11 The *scalar-default-char-expr* shall evaluate to KEEP or DELETE. The STATUS= specifier determines
 12 the disposition of the file that is connected to the specified unit. KEEP shall not be specified for a file
 13 whose status prior to execution of a CLOSE statement is SCRATCH. If KEEP is specified for a file that
 14 exists, the file continues to exist after the execution of a CLOSE statement. If KEEP is specified for a
 15 file that does not exist, the file will not exist after the execution of a CLOSE statement. If DELETE is
 16 specified, the file will not exist after the execution of a CLOSE statement. If this specifier is omitted, the
 17 default value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH,
 18 in which case the default value is DELETE.

19 9.5 Data transfer statements

20 9.5.1 General

- 21 The **READ statement** is the data transfer input statement. The **WRITE statement** and the
 22 **PRINT statement** are the data transfer output statements.

23 R910	<i>read-stmt</i>	is READ (<i>io-control-spec-list</i>) [<i>input-item-list</i>]
24		or READ <i>format</i> [, <i>input-item-list</i>]
25 R911	<i>write-stmt</i>	is WRITE (<i>io-control-spec-list</i>) [<i>output-item-list</i>]
26 R912	<i>print-stmt</i>	is PRINT <i>format</i> [, <i>output-item-list</i>]

NOTE 9.27

Examples of data transfer statements are:

```
READ (6, *) SIZE
READ 10, A, B
WRITE (6, 10) A, S, J
PRINT 10, A, S, J
```

NOTE 9.27 (cont.)

10 FORMAT (2E16.3, I5)

1 9.5.2 Control information list

2 9.5.2.1 Syntax

3 A control information list is an *io-control-spec-list*. It governs data transfer.

4 R913 *io-control-spec* is [UNIT =] *io-unit*
5 or [FMT =] *format*
6 or [NML =] *namelist-group-name*
7 or ADVANCE = *scalar-default-char-expr*
8 or ASYNCHRONOUS = *scalar-char-initialization-expr*
9 or BLANK = *scalar-default-char-expr*
10 or DECIMAL = *scalar-default-char-expr*
11 or DELIM = *scalar-default-char-expr*
12 or END = *label*
13 or EOR = *label*
14 or ERR = *label*
15 or ID = *scalar-int-variable*
16 or IOMSG = *iomsg-variable*
17 or IOSTAT = *scalar-int-variable*
18 or PAD = *scalar-default-char-expr*
19 or POS = *scalar-int-expr*
20 or REC = *scalar-int-expr*
21 or ROUND = *scalar-default-char-expr*
22 or SIGN = *scalar-default-char-expr*
23 or SIZE = *scalar-int-variable*

24 C910 No specifier shall appear more than once in a given *io-control-spec-list*.

25 C911 An *io-unit* shall be specified in an *io-control-spec-list*; if the optional characters UNIT= are
26 omitted, the *io-unit* shall be the first item in the *io-control-spec-list*.

27 C912 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.28 C913 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.

29 C914 (R913) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a
30 branch target statement that appears in the same scoping unit as the data transfer statement.

31 C915 (R913) A *namelist-group-name* shall be the name of a namelist group.

32 C916 (R913) A *namelist-group-name* shall not appear if an *input-item-list* or an *output-item-list*
33 appears in the data transfer statement.

34 C917 (R913) An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.

35 C918 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-*
36 *control-spec-list* and the first item shall be *io-unit*.

37 C919 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item
38 in the *io-control-spec-list* and the first item shall be *io-unit*.

39 C920 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC=

1 specifier or a POS= specifier.

2 C921 (R913) If the REC= specifier appears, an END= specifier shall not appear, a *namelist-group*-
3 *name* shall not appear, and the *format*, if any, shall not be an asterisk.

4 C922 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream in-
5 put/output statement with explicit format specification (10.2) whose control information list
6 does not contain an *internal-file-variable* as the *io-unit*.

7 C923 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.

8 C924 (R913) If a SIZE= specifier appears, an ADVANCE= specifier also shall appear.

9 C925 (R913) The *scalar-char-initialization-expr* in an ASYNCHRONOUS= specifier shall be of type
10 default character and shall have the value YES or NO.

11 C926 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a
12 *file-unit-number*.

13 C927 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall
14 also appear.

15 C928 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.

16 C929 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format*
17 or *namelist-group-name* shall also appear.

18 C930 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name*
19 shall appear.

20 A SIZE= specifier may appear only in an input statement that contains an ADVANCE= specifier with
21 the value NO.

22 An EOR= specifier may appear only in an input statement that contains an ADVANCE= specifier with
23 the value NO.

24 If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a **formatted**
25 **input/output statement**; otherwise, it is an **unformatted input/output statement**.

26 The ADVANCE=, ASYNCHRONOUS=, DECIMAL=, BLANK=, DELIM=, PAD=, SIGN=, and
27 ROUND= specifiers have a limited list of character values. Any trailing blanks are ignored. The
28 values specified are without regard to case.

29 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.

NOTE 9.28

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

30 9.5.2.2 Format specification in a data transfer statement

31 The *format* specifier supplies a format specification or specifies list-directed formatting for a formatted
32 input/output statement.

```

33 R914  format           is default-char-expr
34      or label

```

or *

C931 (R914) The *label* shall be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the FMT= specifier.

The *default-char-expr* shall evaluate to a valid format specification (10.2.1 and 10.2.2).

NOTE 9.29

A *default-char-expr* includes a character constant.

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.

If *format* is *, the statement is a **list-directed input/output statement**.

NOTE 9.30

An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")") X, Y, Z
```

where CHAR_FMT is a default character variable.

9.5.2.3 NML= specifier in a data transfer statement

The NML= specifier supplies the *namelist-group-name* (5.6). This name identifies a particular collection of data objects on which transfer is to be performed.

If a *namelist-group-name* appears, the statement is a **namelist input/output statement**.

9.5.2.4 ADVANCE= specifier in a data transfer statement

The *scalar-default-char-expr* shall evaluate to YES or NO. The ADVANCE= specifier determines whether advancing input/output occurs for a nonchild input/output statement. If YES is specified for a nonchild input/output statement, advancing input/output occurs. If NO is specified, nonadvancing input/output occurs (9.2.3.1). If this specifier is omitted from a nonchild input/output statement that allows the specifier, the default value is YES. A formatted child input/output statement is a nonadvancing input/output statement, and any ADVANCE= specifier is ignored.

9.5.2.5 ASYNCHRONOUS= specifier in a data transfer statement

The ASYNCHRONOUS= specifier determines whether this input/output 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**.

Asynchronous input/output is permitted only for external files opened with an ASYNCHRONOUS= specifier with the value YES in the OPEN statement.

NOTE 9.31

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 9.31 (cont.)

The ASYNCHRONOUS= specifier value in a data transfer statement is an initialization expression because it effects compiler optimizations and, therefore, needs to be known at compile time.

1 The processor may perform an asynchronous data transfer operation asynchronously, but it is not re-
 2 quired to do so. For each external file, records and file storage units read or written by asynchronous
 3 data transfer statements are read, written, and processed in the same order as they would have been if
 4 the data transfer statements were synchronous.

5 If a variable is used in an asynchronous data transfer statement as

- 6 (1) an item in an input/output list,
- 7 (2) a group object in a namelist, or
- 8 (3) a SIZE= specifier

9 the base object of the *data-ref* is implicitly given the ASYNCHRONOUS attribute in the scoping unit
 10 of the data transfer statement. This attribute may be confirmed by explicit declaration.

11 When an asynchronous input/output statement is executed, the set of storage units specified by the
 12 item list or NML= specifier, plus the storage units specified by the SIZE= specifier, is defined to be the
 13 pending input/output storage sequence for the data transfer operation.

NOTE 9.32

A pending input/output storage sequence is not necessarily a contiguous set of storage units.

14 A pending input/output storage sequence **affector** is a variable of which any part is associated with a
 15 storage unit in a pending input/output storage sequence.

16 **9.5.2.6 BLANK= specifier in a data transfer statement**

17 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier temporarily
 18 changes (9.4.1) the blank interpretation mode (10.8.6, 9.4.5.4) for the connection. If the specifier is
 19 omitted, the mode is not changed.

20 **9.5.2.7 DECIMAL= specifier in a data transfer statement**

21 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier temporar-
 22 ily changes (9.4.1) the decimal edit mode (10.6, 10.8.8, 9.4.5.5) for the connection. If the specifier is
 23 omitted, the mode is not changed.

24 **9.5.2.8 DELIM= specifier in a data transfer statement**

25 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= spec-
 26 ifier temporarily changes (9.4.1) the delimiter mode (10.10.4, 10.11.4.1, 9.4.5.6) for the connection. If
 27 the specifier is omitted, the mode is not changed.

28 **9.5.2.9 ID= specifier in a data transfer statement**

29 Successful execution of an asynchronous data transfer statement containing an ID= specifier causes the
 30 variable specified in the ID= specifier to become defined with a processor determined value. This value
 31 is referred to as the identifier of the data transfer operation. It can be used in a subsequent WAIT or
 32 INQUIRE statement to identify the particular data transfer operation.

33 If an error occurs during the execution of a data transfer statement containing an ID= specifier, the
 variable specified in the ID= specifier becomes undefined.

1 A child data transfer statement shall not specify the ID= specifier.

2 **9.5.2.10 PAD= specifier in a data transfer statement**

3 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier temporarily changes
4 (9.4.1) the pad mode (9.5.4.4.2, 9.4.5.11) for the connection. If the specifier is omitted, the mode is not
5 changed.

6 **9.5.2.11 POS= specifier in a data transfer statement**

7 The POS= specifier specifies the file position in file storage units. This specifier may appear in a data
8 transfer statement only if the statement specifies a unit connected for stream access. A child data
9 transfer statement shall not specify this specifier.

10 A processor may prohibit the use of POS= with particular files that do not have the properties necessary
11 to support random positioning. A processor may also prohibit positioning a particular file to any
12 position prior to its current file position if the file does not have the properties necessary to support such
13 positioning.

NOTE 9.33

A unit that is connected to a device or data stream might not be positionable.

14 If the file is connected for formatted stream access, the file position specified by POS= shall be equal to
15 either 1 (the beginning of the file) or a value previously returned by a POS= specifier in an INQUIRE
16 statement for the file.

17 **9.5.2.12 REC= specifier in a data transfer statement**

18 The REC= specifier specifies the number of the record that is to be read or written. This specifier
19 may appear only in an input/output statement that specifies a unit connected for direct access; it
20 shall not appear in a child data transfer statement. If the control information list contains a REC=
21 specifier, the statement is a **direct access input/output statement**. A child data transfer statement
22 is a direct access data transfer statement if the parent is a direct access data transfer statement. Any
23 other data transfer statement is a **sequential access input/output statement** or a **stream access**
24 **input/output statement**, depending on whether the file connection is for sequential access or stream
25 access.

26 **9.5.2.13 ROUND= specifier in a data transfer statement**

27 The *scalar-default-char-expr* shall evaluate to one of the values specified in 9.4.5.14. The ROUND=
28 specifier temporarily changes (9.4.1) the I/O rounding mode (10.7.2.3.7, 9.4.5.14) for the connection. If
29 the specifier is omitted, the mode is not changed.

30 **9.5.2.14 SIGN= specifier in a data transfer statement**

31 The *scalar-default-char-expr* shall evaluate to PLUS, SUPPRESS, or PROCESSOR_DEFINED. The
32 SIGN= specifier temporarily changes (9.4.1) the sign mode (10.8.4, 9.4.5.15) for the connection. If the
33 specifier is omitted, the mode is not changed.

34 **9.5.2.15 SIZE= specifier in a data transfer statement**

35 When a synchronous nonadvancing input statement terminates, the variable specified in the SIZE=
36 specifier becomes defined with the count of the characters transferred by data edit descriptors during
37 execution of the current input statement. Blanks inserted as padding (9.5.4.4.2) are not counted.

- 1 For asynchronous nonadvancing input, the storage units specified in the SIZE= specifier become defined
 2 with the count of the characters transferred when the corresponding wait operation is executed.

3 9.5.3 Data transfer input/output list

- 4 An input/output list specifies the entities whose values are transferred by a data transfer input/output
 5 statement.

6 R915 *input-item* **is** *variable*
 7 **or** *io-implied-do*

8 R916 *output-item* **is** *expr*
 9 **or** *io-implied-do*

10 R917 *io-implied-do* **is** (*io-implied-do-object-list* , *io-implied-do-control*)

11 R918 *io-implied-do-object* **is** *input-item*
 12 **or** *output-item*

13 R919 *io-implied-do-control* **is** *do-variable* = *scalar-int-expr* , ■
 14 ■ *scalar-int-expr* [, *scalar-int-expr*]

15 C932 (R915) A variable that is an *input-item* shall not be a whole assumed-size array.

16 C933 (R915) A variable that is an *input-item* shall not be a procedure pointer.

17 C934 (R919) The *do-variable* shall be a named scalar variable of type integer.

18 C935 (R918) In an *input-item-list*, an *io-implied-do-object* shall be an *input-item*. In an *output-item-*
 19 *list*, an *io-implied-do-object* shall be an *output-item*.

20 C936 (R916) An expression that is an *output-item* shall not have a value that is a procedure pointer.

21 An *input-item* shall not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that
 22 contains the *input-item*.

NOTE 9.34

A constant, an expression involving operators or function references that does not have a pointer result, or an expression enclosed in parentheses shall not appear as an input list item.

- 23 If an input item is a pointer, it shall be associated with a definable target and data are transferred from
 24 the file to the associated target. If an output item is a pointer, it shall be associated with a target and
 25 data are transferred from the target to the file.

NOTE 9.35

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).

- 26 If an input item or an output item is allocatable, it shall be allocated.

- 27 A list item shall not be polymorphic unless it is processed by a user-defined derived-type input/output
 28 procedure (9.5.4.7).

- 29 The *do-variable* of an *io-implied-do* that is in another *io-implied-do* shall not appear as, nor be associated

- 1 with, the *do-variable* of the containing *io-implied-do*.
- 2 The following rules describing whether to expand an input/output list item are re-applied to each
3 expanded list item until none of the rules apply.
- 4 • If an array appears as an input/output list item, it is treated as if the elements, if any, were
5 specified in array element order (6.2.2.1). However, no element of that array may affect the value
6 of any expression in the *input-item*, nor may any element appear more than once in an *input-item*.

NOTE 9.36

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

```

- 7 • If a list item of derived type in an unformatted input/output statement is not processed by a
8 user-defined derived-type input/output procedure (9.5.4.7), and if any subobject of that list item
9 would be processed by a user-defined derived-type input/output procedure, the list item is treated
10 as if all of the components of the object were specified in the list in component order (4.5.4.6);
11 those components shall be accessible in the scoping unit containing the input/output statement
12 and shall not be pointers or allocatable.
- 13 • An effective input/output list item of derived type in an unformatted input/output statement is
14 treated as a single value in a processor-dependent form unless the list item or a subobject thereof
15 is processed by a user-defined derived-type input/output procedure (9.5.4.7).

NOTE 9.37

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.

- 16 • If a list item of derived type in a formatted input/output statement is not processed by a user-
17 defined derived-type input/output procedure, that list item is treated as if all of the components
18 of the list item were specified in the list in component order; those components shall be accessible
19 in the scoping unit containing the input/output statement and shall not be pointers or allocatable.
- 20 • If a derived-type list item is not treated as a list of its individual components, that list item's
21 ultimate components shall not have the **POINTER** or **ALLOCATABLE** attribute unless that list
22 item is processed by a user-defined derived-type input/output procedure.
- 23 • The scalar objects resulting when a data transfer statement's list items are expanded according
24 to the rules in this subclause for handling array and derived-type list items are called **effective**

- 1 **items.** Zero-sized arrays and *io-implied-dos* with an iteration count of zero do not contribute to
 2 the effective list items. A scalar character item of zero length is an effective list item.

NOTE 9.38

In a formatted input/output statement, edit descriptors are associated with effective list items, which are always scalar. The rules in 9.5.3 determine the set of effective list items corresponding to each actual list item in the statement. These rules might have to be applied repetitively until all of the effective list items are scalar items.

- 3 • For an *io-implied-do*, the loop initialization and execution are the same as for a DO construct
 4 (8.1.7.5).

NOTE 9.39

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
```

- 5 • An input/output list shall not contain an item of nondefault character type if the input/output
 6 statement specifies an internal file of default character type. An input/output list shall not con-
 7 tain an item of nondefault character type other than ISO 10646 or ASCII character type if the
 8 input/output statement specifies an internal file of ISO 10646 character type. An input/output
 9 list shall not contain a character item of any character type other than ASCII character type if
 10 the input/output statement specifies an internal file of ASCII character type.

9.5.4 Execution of a data transfer input/output statement

- 11 Execution of a WRITE or PRINT statement for a file that does not exist creates the file unless an error
 12 condition occurs.

- 13 The effect of executing a synchronous data transfer input/output statement shall be as if the following
 14 operations were performed in the order specified.

- 15 (1) Determine the direction of data transfer.
- 16 (2) Identify the unit.
- 17 (3) Perform a wait operation for all pending input/output operations for the unit. If an error,
 18 end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4
 19 through 8 are skipped for the current data transfer statement.
- 20 (4) Establish the format if one is specified.
- 21 (5) If the statement is not a child data transfer statement (9.5.4.7),
 22 (a) position the file prior to data transfer (9.2.3.2), and
 23 (b) for formatted data transfer, set the left tab limit (10.8.1.1).
- 24 (6) Transfer data between the file and the entities specified by the input/output list (if any) or
 25 namelist.
- 26 (7) Determine whether an error, end-of-file, or end-of-record condition has occurred.
- 27 (8) Position the file after data transfer (9.2.3.3) unless the statement is a child data transfer
 28 statement (9.5.4.7).
- 29 (9) Cause any variable specified in a SIZE= specifier to become defined.
- 30 (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified
 31 in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.
- 32 (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified
 33 in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

- 33 The effect of executing an asynchronous data transfer input/output statement shall be as if the following
 34 operations were performed in the order specified.

- 1 (1) Determine the direction of data transfer.
- 2 (2) Identify the unit.
- 3 (3) Establish the format if one is specified.
- 4 (4) Position the file prior to data transfer (9.2.3.2) and, for formatted data transfer, set the left
5 tab limit (10.8.1.1).
- 6 (5) Establish the set of storage units identified by the input/output list. For a READ statement,
7 this might require some or all of the data in the file to be read if an input variable is used
8 as a *scalar-int-expr* in an *io-implied-do-control* in the input/output list, as a *subscript*,
9 *substring-range*, *stride*, or is otherwise referenced.
- 10 (6) Initiate an asynchronous data transfer between the file and the entities specified by the
11 input/output list (if any) or namelist. The asynchronous data transfer may complete (and
12 an error, end-of-file, or end-of-record condition may occur) during the execution of this data
13 transfer statement or during a later wait operation.
- 14 (7) Determine whether an error, end-of-file, or end-of-record condition has occurred. The con-
15 ditions may occur during the execution of this data transfer statement or during the corre-
16 sponding wait operation, but not both.
17 Also, any of these conditions that would have occurred during the corresponding wait oper-
18 ation for a previously pending data transfer operation that does not have an ID= specifier
19 may occur during the execution of this data transfer statement.

J3 internal note

Unresolved Technical Issue 091

Contradiction; two paragraphs earlier, we say it occurs during the execution of the original i/o statement or “during a later wait operation”. There is no wait operation in an async i/o statement. The best fix for this is to probably add an additional action to the list: “Optionally, perform a wait operation for a pending input/output operation for this unit.”

Also, note the difference in specification between this item (para 1) and the text four paragraphs after the list on wait operations. These should be consistent, and preferably specified only once.

- 20 (8) Position the file as if the data transfer had finished (9.2.3.3).
- 21 (9) Cause any variable specified in a SIZE= specifier to become undefined.
- 22 (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified
23 in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

24 For an asynchronous data transfer statement, the data transfers may occur during execution of the
25 statement, during execution of the corresponding wait operation, or anywhere between. The data transfer
26 operation is considered to be pending until a corresponding wait operation is performed.

27 For asynchronous output, a pending input/output storage sequence affector (9.5.2.5) shall not be rede-
28 fined, become undefined, or have its pointer association status changed.

29 For asynchronous input, a pending input/output storage sequence affector shall not be referenced, be-
30 come defined, become undefined, become associated with a dummy argument that has the VALUE
31 attribute, or have its pointer association status changed.

32 Error, end-of-file, and end-of-record conditions in an asynchronous data transfer operation may occur
33 during execution of either the data transfer statement or the corresponding wait operation. If an ID=
34 specifier does not appear in the initiating data transfer statement, the conditions may occur during the
35 execution of any subsequent data transfer or wait operation for the same unit. When a condition occurs
36 for a previously executed asynchronous data transfer statement, a wait operation is performed for all
37 pending data transfer operations on that unit. When a condition occurs during a subsequent statement,
38 any actions specified by IOSTAT=, IOMSG=, ERR=, END=, and EOR= specifiers for that statement
39 are taken.

NOTE 9.40

Because end-of-file and error conditions for asynchronous data transfer statements without an ID= specifier may be reported by the processor during the execution of a subsequent data transfer statement, it may be impossible for the user to determine which input/output statement caused the condition. Reliably detecting which READ statement caused an end-of-file condition requires that all asynchronous READ statements for the unit include an ID= specifier.

1 9.5.4.1 Direction of data transfer

2 Execution of a READ statement causes values to be transferred from a file to the entities specified by
 3 the input list, if any, or specified within the file itself for namelist input. Execution of a WRITE or
 4 PRINT statement causes values to be transferred to a file from the entities specified by the output list
 5 and format specification, if any, or by the *namelist-group-name* for namelist output.

6 9.5.4.2 Identifying a unit

7 A data transfer input/output statement that contains an input/output control list includes a UNIT=
 8 specifier that identifies an external or internal unit. A READ statement that does not contain an
 9 input/output control list specifies a particular processor-dependent unit, which is the same as the unit
 10 identified by * in a READ statement that contains an input/output control list. The PRINT statement
 11 specifies some other processor-dependent unit, which is the same as the unit identified by * in a WRITE
 12 statement. Thus, each data transfer input/output statement identifies an external or internal unit.

13 The unit identified by an unformatted data transfer statement shall be an external unit.

14 The unit identified by a data transfer input/output statement shall be connected to a file when execution
 15 of the statement begins.

NOTE 9.41

The unit may be preconnected.

16 9.5.4.3 Establishing a format

17 If the input/output control list contains * as a format, list-directed formatting is established. If *namelist-*
 18 *group-name* appears, namelist formatting is established. If no *format* or *namelist-group-name* is speci-
 19 fied, unformatted data transfer is established. Otherwise, the format specified by *format* is established.

20 For output to an internal file, a format specification that is in the file or is associated with the file shall
 21 not be specified.

22 9.5.4.4 Data transfer

23 Data are transferred between the file and the entities specified by the input/output list or namelist.
 24 The list items are processed in the order of the input/output list for all data transfer input/output
 25 statements except namelist formatted data transfer statements. The list items for a namelist input
 26 statement are processed in the order of the entities specified within the input records. The list items
 27 for a namelist output statement are processed in the order in which the variables are specified in the
 28 *namelist-group-object-list*. Effective items are derived from the input/output list items as described in
 29 9.5.3.

30 All values needed to determine which entities are specified by an input/output list item are determined
 31 at the beginning of the processing of that item.

32 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 input/output statements.

1

NOTE 9.42

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.

2 All values following the *name=* part of the namelist entity (10.11) within the input records are transmit-
 3 ted to the matching entity specified in the *namelist-group-object-list* prior to processing any succeeding
 4 entity within the input record for namelist input statements. If an entity is specified more than once
 5 within the input record during a namelist formatted data transfer input statement, the last occurrence
 6 of the entity specifies the value or values to be used for that entity.

7 An input list item, or an entity associated with it, shall not contain any portion of an established format
 8 specification.

9 If the input/output item is a pointer, data are transferred between the file and the associated target.

10 If an internal file has been specified, an input/output list item shall not be in the file or associated with
 11 the file.

NOTE 9.43

The file is a data object.

12 A DO variable becomes defined and its iteration count established at the beginning of processing of the
 13 items that constitute the range of an *io-implied-do*.

14 On output, every entity whose value is to be transferred shall be defined.

15 9.5.4.4.1 Unformatted data transfer

16 During unformatted data transfer, data are transferred without editing between the file and the entities
 17 specified by the input/output list. If the file is connected for sequential or direct access, exactly one
 18 record is read or written.

19 A value in the file is stored in a contiguous sequence of file storage units, beginning with the file storage
 20 unit immediately following the current file position.

21 After each value is transferred, the current file position is moved to a point immediately after the last
 22 file storage unit of the value.

23 On input from a file connected for sequential or direct access, the number of file storage units required
 24 by the input list shall be less than or equal to the number of file storage units in the record.

25 On input, if the file storage units transferred do not contain a value with the same type and type
 26 parameters as the input list entity, then the resulting value of the entity is processor-dependent except
 27 in the following cases.

- 28 (1) A complex entity may correspond to two real values with the same kind type parameter as
 29 the complex entity.
- 30 (2) A default character list entity of length *n* may correspond to *n* default characters stored in
 31 the file, regardless of the length parameters of the entities that were written to these storage
 32 units of the file. If the file is connected for stream input, the characters may have been
 33 written by formatted stream output.

1 On output to a file connected for unformatted direct access, the output list shall not specify more values
2 than can fit into the record. If the file is connected for direct access and the values specified by the
3 output list do not fill the record, the remainder of the record is undefined.

4 If the file is connected for unformatted sequential access, the record is created with a length sufficient
5 to hold the values from the output list. This length shall be one of the set of allowed record lengths for
6 the file and shall not exceed the value specified in the RECL= specifier, if any, of the OPEN statement
7 that established the connection.

8 If the file is not connected for unformatted input/output, unformatted data transfer is prohibited.

9 **9.5.4.4.2 Formatted data transfer**

10 During formatted data transfer, data are transferred with editing between the file and the entities
11 specified by the input/output list or by the *namelist-group-name*. Format control is initiated and editing
12 is performed as described in Clause 10.

13 The current record and possibly additional records are read or written.

14 If the file is not connected for formatted input/output, formatted data transfer is prohibited.

15 During advancing input when the pad mode has the value NO, the input list and format specification
16 shall not require more characters from the record than the record contains.

17 During advancing input when the pad mode has the value YES, blank characters are supplied by the
18 processor if the input list and format specification require more characters from the record than the
19 record contains.

20 During nonadvancing input when the pad mode has the value NO, an end-of-record condition (9.10)
21 occurs if the input list and format specification require more characters from the record than the record
22 contains, and the record is complete (9.2.2.3). If the record is incomplete, an end-of-file condition occurs
23 instead of an end-of-record condition.

24 During nonadvancing input when the pad mode has the value YES, blank characters are supplied by
25 the processor if an effective item and its corresponding data edit descriptors require more characters
26 from the record than the record contains. If the record is incomplete, an end-of-file condition occurs;
27 otherwise an end-of-record condition occurs.

28 If the file is connected for direct access, the record number is increased by one as each succeeding record
29 is read or written.

30 On output, if the file is connected for direct access or is an internal file and the characters specified by
31 the output list and format do not fill a record, blank characters are added to fill the record.

32 On output, the output list and format specification shall not specify more characters for a record than
33 have been specified by a RECL= specifier in the OPEN statement or the record length of an internal
34 file.

35 **9.5.4.5 List-directed formatting**

36 If list-directed formatting has been established, editing is performed as described in 10.10.

37 **9.5.4.6 Namelist formatting**

38 If namelist formatting has been established, editing is performed as described in 10.11.

1 Every allocatable *namelist-group-object* in the namelist group shall be allocated and every *namelist-*
 2 *group-object* that is a pointer shall be associated with a target. If a *namelist-group-object* is polymorphic
 3 or has an ultimate component that is allocatable or a pointer, that object shall be processed by a user-
 4 defined derived-type input/output procedure (9.5.4.7).

5 **9.5.4.7 User-defined derived-type input/output**

6 User-defined derived-type input/output procedures allow a program to override the default handling of
 7 derived-type objects and values in data transfer input/output statements described in 9.5.3.

8 A user-defined derived-type input/output procedure is a procedure accessible by a *dtio-generic-spec*
 9 (12.4.3.2). A particular user-defined derived-type input/output procedure is selected as described in
 10 9.5.4.7.3.

11 **9.5.4.7.1 Executing user-defined derived-type input/output data transfers**

12 If a derived-type input/output procedure is selected as specified in 9.5.4.7.3, the processor shall call the se-
 13 lected user-defined derived-type input/output procedure for any appropriate data transfer input/output
 14 statements executed in that scoping unit. The user-defined derived-type input/output procedure controls
 15 the actual data transfer operations for the derived-type list item.

16 A data transfer statement that includes a derived-type list item and that causes a user-defined derived-
 17 type input/output procedure to be invoked is called a **parent data transfer statement**. A data
 18 transfer statement that is executed while a parent data transfer statement is being processed and that
 19 specifies the unit passed into a user-defined derived-type input/output procedure is called a **child data**
 20 **transfer statement**.

NOTE 9.44

A user-defined derived-type input/output procedure will usually contain child data transfer state-
 ments that read values from or write values to the current record or at the current file position.
 The effect of executing the user-defined derived-type input/output procedure is similar to that of
 substituting the list items from any child data transfer statements into the parent data transfer
 statement's list items, along with similar substitutions in the format specification.

NOTE 9.45

A particular execution of a READ, WRITE or PRINT statement can be both a parent and a
 child data transfer statement. A user-defined derived-type input/output procedure can indirectly
 call itself or another user-defined derived-type 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 user-defined derived-type input/output procedure calls itself indirectly
 in this manner, it shall be declared RECURSIVE.

21 A child data transfer statement is processed differently from a nonchild data transfer statement in the
 22 following ways.

- 23 • Executing a child data transfer statement does not position the file prior to data transfer.
- 24 • An unformatted child data transfer statement does not position the file after data transfer is
 25 complete.
- 26 • Any ADVANCE= specifier in a child input/output statement is ignored.

27 **9.5.4.7.2 User-defined derived-type input/output procedures**

1 For a particular derived type and a particular set of kind type parameter values, there are four possible
 2 sets of characteristics for user-defined derived-type input/output procedures; one each for formatted
 3 input, formatted output, unformatted input, and unformatted output. The user need not supply all four
 4 procedures. The procedures are specified to be used for derived-type input/output by interface blocks
 5 (12.4.3.2) or by generic bindings (4.5.5), with a *dtio-generic-spec* (R1208).

6 In the four interfaces, which specify the characteristics of user-defined procedures for derived-type in-
 7 put/output, the following syntax term is used:

```
8 R920   dtv-type-spec                is TYPE( derived-type-spec )
9                                     or CLASS( derived-type-spec )
```

10 C937 (R920) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used;
 11 otherwise, the TYPE keyword shall be used.

12 C938 (R920) All length type parameters of *derived-type-spec* shall be assumed.

13 If the *dtio-generic-spec* is READ (FORMATTED), the characteristics shall be the same as those specified
 14 by the following interface:

```
15         SUBROUTINE my_read_routine_formatted                &
16                                     (dtv,                    &
17                                     unit,                    &
18                                     iotype, v_list,          &
19                                     iostat, iomsg)
20         ! the derived-type variable
21         dtv-type-spec, INTENT(INOUT) :: dtv
22         INTEGER, INTENT(IN) :: unit ! unit number
23         ! the edit descriptor string
24         CHARACTER (LEN=*), INTENT(IN) :: iotype
25         INTEGER, INTENT(IN) :: v_list(:)
26         INTEGER, INTENT(OUT) :: iostat
27         CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
28         END
```

29 If the *dtio-generic-spec* is READ (UNFORMATTED), the characteristics shall be the same as those
 30 specified by the following interface:

```
31         SUBROUTINE my_read_routine_unformatted            &
32                                     (dtv,                    &
33                                     unit,                    &
34                                     iostat, iomsg)
35         ! the derived-type variable
36         dtv-type-spec, INTENT(INOUT) :: dtv
37         INTEGER, INTENT(IN) :: unit
38         INTEGER, INTENT(OUT) :: iostat
39         CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
40         END
```

41 If the *dtio-generic-spec* is WRITE (FORMATTED), the characteristics shall be the same as those spec-
 42 ified by the following interface:

```
        SUBROUTINE my_write_routine_formatted                &
```

```

1
2                (dtv,                &
3                unit,                &
4                iotype, v_list,      &
5                iostat, iomsg)
6        ! the derived-type value/variable
7        dtv-type-spec, INTENT(IN) :: dtv
8        INTEGER, INTENT(IN) :: unit
9        ! the edit descriptor string
10       CHARACTER (LEN=*), INTENT(IN) :: iotype
11       INTEGER, INTENT(IN) :: v_list(:)
12       INTEGER, INTENT(OUT) :: iostat
13       CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
14       END

```

15 If the *dtio-generic-spec* is WRITE (UNFORMATTED), the characteristics shall be the same as those
 16 specified by the following interface:

```

17       SUBROUTINE my_write_routine_unformatted      &
18                (dtv,                &
19                unit,                &
20                iostat, iomsg)
21        ! the derived-type value/variable
22        dtv-type-spec, INTENT(IN) :: dtv
23        INTEGER, INTENT(IN) :: unit
24        INTEGER, INTENT(OUT) :: iostat
25        CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
26       END

```

27 The actual specific procedure names (the *my_..._routine_...* procedure names above) are not signifi-
 28 cant. In the discussion here and elsewhere, the dummy arguments in these interfaces are referred by the
 29 names given above; the names are, however, arbitrary.

30 When a user-defined derived-type input/output procedure is invoked, the processor shall pass a *unit*
 31 argument that has a value as follows.

- 32 • If the parent data transfer statement uses a *file-unit-number*, the value of the *unit* argument shall
 33 be that of the *file-unit-number*.
- 34 • If the parent data transfer statement is a WRITE statement with an asterisk unit or a PRINT
 35 statement, the *unit* argument shall have the same value as the OUTPUT_UNIT named constant
 36 of the ISO_FORTRAN_ENV intrinsic module (13.8.2).
- 37 • If the parent data transfer statement is a READ statement with an asterisk unit or a READ
 38 statement without an *io-control-spec-list*, the *unit* argument shall have the same value as the
 39 INPUT_UNIT named constant of the ISO_FORTRAN_ENV intrinsic module (13.8.2).
- 40 • Otherwise the parent data transfer statement must access an internal file, in which case the *unit*
 41 argument shall have a processor-dependent negative value.

NOTE 9.46

The **unit** argument passed to a user-defined derived-type input/output procedure will be negative when the parent input/output 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 **ISO_FORTRAN_ENV** intrinsic module (13.8.2).

- 1 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 edit descriptor, if the parent data
- 5 transfer statement contained a format specification and the list item’s corresponding edit descriptor
- 6 was a **DT** edit descriptor.
- 7 If the parent data transfer statement is a **READ** statement, the **dtv** dummy argument is argument
- 8 associated with the effective list item that caused the user-defined derived-type input procedure to be
- 9 invoked, as if the effective list item were an actual argument in this procedure reference (2.5.6).
- 10 If the parent data transfer statement is a **WRITE** or **PRINT** statement, the processor shall provide the
- 11 value of the effective list item in the **dtv** dummy argument.
- 12 If the *v-list* of the edit descriptor appears in the parent data transfer statement, the processor shall
- 13 provide the values from it in the **v_list** dummy argument, with the same number of elements in the
- 14 same order as *v-list*. If there is no *v-list* in the edit descriptor or if the data transfer statement specifies
- 15 list-directed or namelist formatting, the processor shall provide **v_list** as a zero-sized array.

NOTE 9.47

The user’s procedure may 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.

- 16 The **iostat** argument is used to report whether an error, end-of-record, or end-of-file condition (9.10)
- 17 occurs. If an error condition occurs, the user-defined derived-type input/output procedure shall assign
- 18 a positive value to the **iostat** argument. Otherwise, if an end-of-file condition occurs, the user-defined
- 19 derived-type input procedure shall assign the value of the named constant **IOSTAT_END** (13.8.2.10)
- 20 to the **iostat** argument. Otherwise, if an end-of-record condition occurs, the user-defined derived-
- 21 type input procedure shall assign the value of the named constant **IOSTAT_EOR** (13.8.2.11) to **iostat**.
- 22 Otherwise, the user-defined derived-type input/output procedure shall assign the value zero to the
- 23 **iostat** argument.
- 24 If the user-defined derived-type input/output procedure returns a nonzero value for the **iostat** argument,
- 25 the procedure shall also return an explanatory message in the **iomsg** argument. Otherwise, the procedure
- 26 shall not change the value of the **iomsg** argument.

NOTE 9.48

The values of the **iostat** and **iomsg** arguments set in a user-defined derived-type input/output procedure need not be passed to all of the parent data transfer statements.

- 27 If the **iostat** argument of the user-defined derived-type input/output procedure has a nonzero value
- 28 when that procedure returns, and the processor therefore terminates execution of the program as de-

- 1 scribed in 9.10, the processor shall make the value of the `iomsg` argument available in a processor-
 2 dependent manner.
- 3 When a parent READ statement is active, an input/output statement shall not read from any external
 4 unit other than the one specified by the dummy argument `unit` and shall not perform output to any
 5 external unit.
- 6 When a parent WRITE or PRINT statement is active, an input/output statement shall not perform
 7 output to any external unit other than the one specified by the dummy argument `unit` and shall not
 8 read from any external unit.
- 9 When a parent data transfer statement is active, a data transfer statement that specifies an internal file
 10 is permitted.
- 11 OPEN, CLOSE, BACKSPACE, ENDFILE, and REWIND statements shall not be executed while a
 12 parent data transfer statement is active.
- 13 A user-defined derived-type input/output procedure may use a FORMAT with a DT edit descriptor for
 14 handling a component of the derived type that is itself of a derived type. A child data transfer statement
 15 that is a list directed or namelist input/output statement may contain a list item of derived type.
- 16 Because a child data transfer statement does not position the file prior to data transfer, the child data
 17 transfer statement starts transferring data from where the file was positioned by the parent data transfer
 18 statement's most recently processed effective list item or record positioning edit descriptor. This is not
 19 necessarily at the beginning of a record.
- 20 A record positioning edit descriptor, such as TL and TR, used on `unit` by a child data transfer statement
 21 shall not cause the record position to be positioned before the record position at the time the user-defined
 22 derived-type input/output procedure was invoked.

NOTE 9.49

A robust user-defined derived-type input/output procedure may wish to use INQUIRE to determine the settings of BLANK=, PAD=, ROUND=, DECIMAL=, and DELIM= for an external unit. The INQUIRE provides values as specified in 9.9.

- 23 Neither a parent nor child data transfer statement shall be asynchronous.
- 24 A user-defined derived-type input/output procedure, and any procedures invoked therefrom, shall not
 25 define, nor cause to become undefined, any storage location referenced by any input/output list item,
 26 the corresponding format, or any specifier in any active parent data transfer statement, except through
 27 the `dtv` argument.

NOTE 9.50

A child data transfer statement shall not specify the ID=, POS=, or REC= specifiers in an input/output control list.

NOTE 9.51

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
```

NOTE 9.51 (cont.)

```

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
    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
  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 9.52

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

```

NOTE 9.52 (cont.)

```

    INTEGER :: value = 0
    TYPE (NODE), POINTER :: next_node => NULL ( )
CONTAINS
    PROCEDURE,PRIVATE :: pwf
    GENERIC             :: WRITE(FORMATTED) => pwf
END TYPE node

CONTAINS

    RECURSIVE 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

```

1 9.5.4.7.3 Resolving derived-type input/output procedure references

2 A suitable generic interface for user-defined derived-type input/output of an effective item is one that
 3 has a *dtio-generic-spec* that is appropriate to the direction (read or write) and form (formatted or
 4 unformatted) of the data transfer as specified in 9.5.4.7, and has a specific interface whose *dtv* argument
 5 is compatible with the effective item according to the rules for argument association in 12.5.2.5.

6 When an effective item (9.5.3) that is of derived-type is encountered during a data transfer, user-defined
 7 derived-type input/output occurs if both of the following conditions are true.

- 8 (1) The circumstances of the input/output are such that user-defined derived-type input/output
 9 is permitted; that is, either
 - 10 (a) the transfer was initiated by a list-directed, namelist, or unformatted input/output
 11 statement, or
 - 12 (b) a format specification is supplied for the input/output statement, and the edit de-
 13 scriptor corresponding to the effective item is a DT edit descriptor.
- 14 (2) A suitable user-defined derived-type input/output procedure is available; that is, either
 - 15 (a) the declared type of the effective item has a suitable generic type-bound procedure,
 16 or
 - 17 (b) a suitable generic interface is accessible.

18 If (2a) is true, the procedure referenced is determined as for explicit type-bound procedure references
 19 (12.5); that is, the binding with the appropriate specific interface is located in the declared type of the
 20 effective item, and the corresponding binding in the dynamic type of the effective item is selected.

21 If (2a) is false and (2b) is true, the reference is to the procedure identified by the appropriate specific

1 interface in the interface block. This reference shall not be to a dummy procedure that is not present,
 2 or to a disassociated procedure pointer.

3 9.5.5 Termination of data transfer statements

4 Termination of an input/output data transfer statement occurs when

- 5 (1) format processing encounters a data edit descriptor and there are no remaining elements in
 6 the *input-item-list* or *output-item-list*,
- 7 (2) unformatted or list-directed data transfer exhausts the *input-item-list* or *output-item-list*,
- 8 (3) namelist output exhausts the *namelist-group-object-list*,
- 9 (4) an error condition occurs,
- 10 (5) an end-of-file condition occurs,
- 11 (6) a slash (/) is encountered as a value separator (10.10, 10.11) in the record being read during
 12 list-directed or namelist input, or
- 13 (7) an end-of-record condition occurs during execution of a nonadvancing input statement
 14 (9.10).

15 9.6 Waiting on pending data transfer

16 9.6.1 Wait operation

17 Execution of an asynchronous data transfer statement in which neither an error, end-of-record, nor end-
 18 of-file condition occurs initiates a pending data transfer operation. There may be multiple pending data
 19 transfer operations for the same or multiple units simultaneously. A pending data transfer operation
 20 remains pending until a corresponding wait operation is performed. A wait operation may be performed
 21 by a WAIT, INQUIRE, FLUSH, CLOSE, data transfer, or file positioning statement.

22 A **wait operation** completes the processing of a pending data transfer operation. Each wait operation
 23 completes only a single data transfer operation, although a single statement may perform multiple wait
 24 operations.

25 If the actual data transfer is not yet complete, the wait operation first waits for its completion. If the
 26 data transfer operation is an input operation that completed without error, the storage units of the
 27 input/output storage sequence then become defined with the values as described in 9.5.2.15 and 9.5.4.4.

28 If any error, end-of-file, or end-of-record conditions occur, the applicable actions specified by the IO-
 29 STAT=, IOMSG=, ERR=, END=, and EOR= specifiers of the statement that performs the wait oper-
 30 ation are taken.

31 If an error or end-of-file condition occurs during a wait operation for a unit, the processor performs a
 32 wait operation for all pending data transfer operations for that unit.

NOTE 9.53

Error, end-of-file, and end-of-record conditions may be raised either during the data transfer state-
 ment that initiates asynchronous input/output, a subsequent asynchronous data transfer statement
 for the same unit, or during the wait operation. If such conditions are raised during a data transfer
 statement, they trigger actions according to the IOSTAT=, ERR=, END=, and EOR= specifiers
 of that statement; if they are raised during the wait operation, the actions are in accordance with
 the specifiers of the statement that performs the wait operation.

33 After completion of the wait operation, the data transfer operation and its input/output storage sequence
 34 are no longer considered to be pending.

1 9.6.2 WAIT statement

2 A **WAIT statement** performs a wait operation for specified pending asynchronous data transfer oper-
3 ations.

NOTE 9.54

The CLOSE, INQUIRE, and file positioning statements may also perform wait operations.

4 R921 *wait-stmt* **is** WAIT (*wait-spec-list*)

5 R922 *wait-spec* **is** [UNIT =] *file-unit-number*

6 **or** END = *label*

7 **or** EOR = *label*

8 **or** ERR = *label*

9 **or** ID = *scalar-int-expr*

10 **or** IOMSG = *iomsg-variable*

11 **or** IOSTAT = *scalar-int-variable*

12 C939 No specifier shall appear more than once in a given *wait-spec-list*.

13 C940 A *file-unit-number* shall be specified in a *wait-spec-list*; if the optional characters UNIT= are
14 omitted, the *file-unit-number* shall be the first item in the *wait-spec-list*.

15 C941 (R922) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a
16 branch target statement that appears in the same scoping unit as the WAIT statement.

17 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.

18 The value of the expression specified in the ID= specifier shall be the identifier of a pending data transfer
19 operation for the specified unit. If the ID= specifier appears, a wait operation for the specified data
20 transfer operation is performed. If the ID= specifier is omitted, wait operations for all pending data
21 transfers for the specified unit are performed.

22 Execution of a WAIT statement specifying a unit that does not exist, has no file connected to it, or is
23 not open for asynchronous input/output is permitted, provided that the WAIT statement has no ID=
24 specifier; such a WAIT statement does not cause an error or end-of-file condition to occur.

NOTE 9.55

An EOR= specifier has no effect if the pending data transfer operation is not a nonadvancing read.
And END= specifier has no effect if the pending data transfer operation is not a READ.

25 9.7 File positioning statements

26 9.7.1 Syntax

27 R923 *backspace-stmt* **is** BACKSPACE *file-unit-number*
28 **or** BACKSPACE (*position-spec-list*)

29 R924 *endfile-stmt* **is** ENDFILE *file-unit-number*
30 **or** ENDFILE (*position-spec-list*)

31 R925 *rewind-stmt* **is** REWIND *file-unit-number*
32 **or** REWIND (*position-spec-list*)

33 A unit that is connected for direct access shall not be referred to by a BACKSPACE, ENDFILE, or

1 REWIND statement. A unit that is connected for unformatted stream access shall not be referred to
 2 by a BACKSPACE statement. A unit that is connected with an ACTION= specifier having the value
 3 READ shall not be referred to by an ENDFILE statement. A unit whose connect team has more than
 4 one image shall not be referred to by a BACKSPACE, ENDFILE, or REWIND statement.

5 R926 *position-spec* is [UNIT =] *file-unit-number*
 6 or IOMSG = *iomsg-variable*
 7 or IOSTAT = *scalar-int-variable*
 8 or ERR = *label*

9 C942 No specifier shall appear more than once in a given *position-spec-list*.

10 C943 A *file-unit-number* shall be specified in a *position-spec-list*; if the optional characters UNIT=
 11 are omitted, the *file-unit-number* shall be the first item in the *position-spec-list*.

12 C944 (R926) The *label* in the ERR= specifier shall be the statement label of a branch target statement
 13 that appears in the same scoping unit as the file positioning statement.

14 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

15 Execution of a file positioning statement performs a wait operation for all pending asynchronous data
 16 transfer operations for the specified unit.

17 9.7.2 BACKSPACE statement

18 Execution of a **BACKSPACE statement** causes the file connected to the specified unit to be positioned
 19 before the current record if there is a current record, or before the preceding record if there is no current
 20 record. If the file is at its initial point, the position of the file is not changed.

NOTE 9.56

If the preceding record is an endfile record, the file is positioned before the endfile record.

21 If a BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before
 22 the record that precedes the endfile record.

23 Backspacing a file that is connected but does not exist is prohibited.

24 Backspacing over records written using list-directed or namelist formatting is prohibited.

NOTE 9.57

An example of a BACKSPACE statement is:

```
BACKSPACE (10, IOSTAT = N)
```

25 9.7.3 ENDFILE statement

26 Execution of an **ENDFILE statement** for a file connected for sequential access writes an endfile record
 27 as the next record of the file. The file is then positioned after the endfile record, which becomes the last
 28 record of the file. If the file also may be connected for direct access, only those records before the endfile
 29 record are considered to have been written. Thus, only those records may be read during subsequent
 30 direct access connections to the file.

31 After execution of an ENDFILE statement for a file connected for sequential access, a BACKSPACE
 32 or REWIND statement shall be used to reposition the file prior to execution of any data transfer
 33 input/output statement or ENDFILE statement.

- 1 Execution of an ENDFILE statement for a file connected for stream access causes the terminal point of
 2 the file to become equal to the current file position. Only file storage units before the current position are
 3 considered to have been written; thus only those file storage units may be subsequently read. Subsequent
 4 stream output statements may be used to write further data to the file.
- 5 Execution of an ENDFILE statement for a file that is connected but does not exist creates the file; if
 6 the file is connected for sequential access, it is created prior to writing the endfile record.

NOTE 9.58

An example of an ENDFILE statement is:

```
ENDFILE K
```

7 9.7.4 REWIND statement

- 8 Execution of a **REWIND statement** causes the specified file to be positioned at its initial point.

NOTE 9.59

If the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

- 9 Execution of a REWIND statement for a file that is connected but does not exist is permitted and has
 10 no effect on any file.

NOTE 9.60

An example of a REWIND statement is:

```
REWIND 10
```

11 9.8 FLUSH statement

- 12 R927 *flush-stmt* **is** FLUSH *file-unit-number*
 13 **or** FLUSH (*flush-spec-list*)

- 14 R928 *flush-spec* **is** [UNIT =] *file-unit-number*
 15 **or** IOSTAT = *scalar-int-variable*
 16 **or** IOMSG = *iomsg-variable*
 17 **or** ERR = *label*

- 18 C945 No specifier shall appear more than once in a given *flush-spec-list*.

- 19 C946 A *file-unit-number* shall be specified in a *flush-spec-list*; if the optional characters UNIT= are
 20 omitted from the unit specifier, the *file-unit-number* shall be the first item in the *flush-spec-list*.

- 21 C947 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement
 22 that appears in the same scoping unit as the FLUSH statement.

- 23 The IOSTAT=, IOMSG= and ERR= specifiers are described in 9.10. The IOSTAT= variable shall
 24 be set to a processor-dependent positive value if an error occurs, to zero if the processor-dependent
 25 flush operation was successful, or to a processor-dependent negative value if the flush operation is not
 26 supported for the unit specified.

- 27 Execution of a **FLUSH statement** causes data written to an external unit to be made available to

- 1 other images of the unit's connect team which execute a FLUSH statement in a subsequent segment for
 2 that unit. It also causes data written to an external file to be available to other processes, or causes
 3 data placed in an external file by means other than Fortran to be available to a READ statement. The
 4 action is processor dependent.

J3 internal note

Unresolved Technical Issue 106

FLUSH action contradiction.

The last sentence contradicts the entire rest of the paragraph it's in.

Well, what is it – is the action processor-defined, or does it cause data written to an external file to be available to other processors and cause data placed in an external file by means other than Fortran to be available to a READ statement?

Hmm, the first half sounds like something outside the scope of the standard anyway, so it could probably be more cautiously worded.

- 5 Execution of a FLUSH statement for a file that is connected but does not exist is permitted and has no
 6 effect on any file. A FLUSH statement has no effect on file position.
- 7 Execution of a FLUSH statement performs a wait operation for all pending asynchronous data transfer
 8 operations for the specified unit.

NOTE 9.61

Because this standard does not specify the mechanism of file storage, the exact meaning of the flush operation is not precisely defined. The intention is that the flush operation should 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 I/O buffers”.

NOTE 9.62

An example of a FLUSH statement is:

```
FLUSH( 10, IOSTAT=N)
```

9.9 File inquiry statement

9.9.1 Forms of the INQUIRE statement

- 11 The **INQUIRE statement** may be used to inquire about properties of a particular named file or of
 12 the connection to a particular unit. There are three forms of the INQUIRE statement: **inquire by file**,
 13 which uses the FILE= specifier, **inquire by unit**, which uses the UNIT= specifier, and **inquire by**
 14 **output list**, which uses only the IOLENGTH= specifier. All specifier value assignments are performed
 15 according to the rules for assignment statements.

- 16 For inquiry by unit, the unit specified need not exist or be connected to a file. If it is connected to a
 17 file, the inquiry is being made about the connection and about the file connected.

- 18 An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values
 19 assigned by an INQUIRE statement are those that are current at the time the statement is executed.

- 20 R929 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
 21 **or** INQUIRE (IOLENGTH = *scalar-int-variable*) ■
 22 ■ *output-item-list*

NOTE 9.63

Examples of INQUIRE statements are:

```
INQUIRE (IOLENGTH = IOL) A (1:N)
INQUIRE (UNIT = JOAN, OPENED = LOG_01, NAMED = LOG_02, &
        FORM = CHAR_VAR, IOSTAT = IOS)
```

9.9.2 Inquiry specifiers

Unless constrained, the following inquiry specifiers may be used in either of the inquire by file or inquire by unit forms of the INQUIRE statement:

R930	<i>inquire-spec</i>	is	[UNIT =] <i>file-unit-number</i>
		or	FILE = <i>file-name-expr</i>
		or	ACCESS = <i>scalar-default-char-variable</i>
		or	ACTION = <i>scalar-default-char-variable</i>
		or	ASYNCHRONOUS = <i>scalar-default-char-variable</i>
		or	BLANK = <i>scalar-default-char-variable</i>
		or	DECIMAL = <i>scalar-default-char-variable</i>
		or	DELIM = <i>scalar-default-char-variable</i>
		or	DIRECT = <i>scalar-default-char-variable</i>
		or	ENCODING = <i>scalar-default-char-variable</i>
		or	ERR = <i>label</i>
		or	EXIST = <i>scalar-default-logical-variable</i>
		or	FORM = <i>scalar-default-char-variable</i>
		or	FORMATTED = <i>scalar-default-char-variable</i>
		or	ID = <i>scalar-int-expr</i>
		or	IOMSG = <i>iomsg-variable</i>
		or	IOSTAT = <i>scalar-int-variable</i>
		or	NAME = <i>scalar-default-char-variable</i>
		or	NAMED = <i>scalar-default-logical-variable</i>
		or	NEXTREC = <i>scalar-int-variable</i>
		or	NUMBER = <i>scalar-int-variable</i>
		or	OPENED = <i>scalar-default-logical-variable</i>
		or	PAD = <i>scalar-default-char-variable</i>
		or	PENDING = <i>scalar-default-logical-variable</i>
		or	POS = <i>scalar-int-variable</i>
		or	POSITION = <i>scalar-default-char-variable</i>
		or	READ = <i>scalar-default-char-variable</i>
		or	READWRITE = <i>scalar-default-char-variable</i>
		or	RECL = <i>scalar-int-variable</i>
		or	ROUND = <i>scalar-default-char-variable</i>
		or	SEQUENTIAL = <i>scalar-default-char-variable</i>
		or	SIGN = <i>scalar-default-char-variable</i>
		or	SIZE = <i>scalar-int-variable</i>
		or	STREAM = <i>scalar-default-char-variable</i>
		or	TEAM = <i>image-team</i>
		or	UNFORMATTED = <i>scalar-default-char-variable</i>
		or	WRITE = <i>scalar-default-char-variable</i>

C948 No specifier shall appear more than once in a given *inquire-spec-list*.

C949 An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not both.

1 C950 In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are
2 omitted, the *file-unit-number* shall be the first item in the *inquire-spec-list*.

3 C951 If an ID= specifier appears in an *inquire-spec-list*, a PENDING= specifier shall also appear.

4 C952 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement
5 that appears in the same scoping unit as the INQUIRE statement.

6 If *file-unit-number* identifies an internal unit (9.5.4.7.2), an error condition occurs.

7 When a returned value of a specifier other than the NAME= specifier is of type character, the value
8 returned is in upper case.

9 If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier
10 variables become undefined, except for variables in the IOSTAT= and IOMSG= specifiers (if any).

11 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

12 9.9.2.1 FILE= specifier in the INQUIRE statement

13 The value of the *file-name-expr* in the FILE= specifier specifies the name of the file being inquired about.
14 The named file need not exist or be connected to a unit. The value of the *file-name-expr* shall be of a
15 form acceptable to the processor as a file name. Any trailing blanks are ignored. The interpretation of
16 case is processor dependent.

17 9.9.2.2 ACCESS= specifier in the INQUIRE statement

18 The *scalar-default-char-variable* in the ACCESS= specifier is assigned the value SEQUENTIAL if the
19 connection is for sequential access, DIRECT if the connection is for direct access, or STREAM if the
20 connection is for stream access. If there is no connection, it is assigned the value UNDEFINED.

21 9.9.2.3 ACTION= specifier in the INQUIRE statement

22 The *scalar-default-char-variable* in the ACTION= specifier is assigned the value READ if the connection
23 is for input only, WRITE if the connection is for output only, and READWRITE if the connection is for
24 both input and output. If there is no connection, the *scalar-default-char-variable* is assigned the value
25 UNDEFINED.

26 9.9.2.4 ASYNCHRONOUS= specifier in the INQUIRE statement

27 The *scalar-default-char-variable* in the ASYNCHRONOUS= specifier is assigned the value YES if the
28 connection allows asynchronous input/output; it is assigned the value NO if the connection does not
29 allow asynchronous input/output. If there is no connection, the *scalar-default-char-variable* is assigned
30 the value UNDEFINED.

31 9.9.2.5 BLANK= specifier in the INQUIRE statement

32 The *scalar-default-char-variable* in the BLANK= specifier is assigned the value ZERO or NULL, corre-
33 sponding to the blank interpretation mode in effect for a connection for formatted input/output. If there
34 is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable*
35 is assigned the value UNDEFINED.

36 9.9.2.6 DECIMAL= specifier in the INQUIRE statement

37 The *scalar-default-char-variable* in the DECIMAL= specifier is assigned the value COMMA or POINT,
38 corresponding to the decimal edit mode in effect for a connection for formatted input/output. If there

1 is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable*
2 is assigned the value UNDEFINED.

3 **9.9.2.7 DELIM= specifier in the INQUIRE statement**

4 The *scalar-default-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE, QUOTE,
5 or NONE, corresponding to the delimiter mode in effect for a connection for formatted input/output.
6 If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-*
7 *variable* is assigned the value UNDEFINED.

8 **9.9.2.8 DIRECT= specifier in the INQUIRE statement**

9 The *scalar-default-char-variable* in the DIRECT= specifier is assigned the value YES if DIRECT is
10 included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of
11 allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether
12 DIRECT is included in the set of allowed access methods for the file.

13 **9.9.2.9 ENCODING= specifier in the INQUIRE statement**

14 The *scalar-default-char-variable* in the ENCODING= specifier is assigned the value UTF-8 if the con-
15 nection is for formatted input/output with an encoding form of UTF-8, and is assigned the value UN-
16 DEFINED if the connection is for unformatted input/output. If there is no connection, it is assigned
17 the value UTF-8 if the processor is able to determine that the encoding form of the file is UTF-8; if
18 the processor is unable to determine the encoding form of the file, the variable is assigned the value
19 UNKNOWN.

NOTE 9.64

The value assigned may be something other than UTF-8, UNDEFINED, or UNKNOWN if the processor supports other specific encoding forms (e.g. UTF-16BE).

20 **9.9.2.10 EXIST= specifier in the INQUIRE statement**

21 Execution of an INQUIRE by file statement causes the *scalar-default-logical-variable* in the EXIST=
22 specifier to be assigned the value true if there exists a file with the specified name; otherwise, false is
23 assigned. Execution of an INQUIRE by unit statement causes true to be assigned if the specified unit
24 exists; otherwise, false is assigned.

25 **9.9.2.11 FORM= specifier in the INQUIRE statement**

26 The *scalar-default-char-variable* in the FORM= specifier is assigned the value FORMATTED if the
27 connection is for formatted input/output, and is assigned the value UNFORMATTED if the connection
28 is for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

29 **9.9.2.12 FORMATTED= specifier in the INQUIRE statement**

30 The *scalar-default-char-variable* in the FORMATTED= specifier is assigned the value YES if FOR-
31 MATTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in
32 the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether
33 FORMATTED is included in the set of allowed forms for the file.

34 **9.9.2.13 ID= specifier in the INQUIRE statement**

35 The value of the expression specified in the ID= specifier shall be the identifier of a pending data transfer
36 operation for the specified unit. This specifier interacts with the PENDING= specifier (9.9.2.20).

1 **9.9.2.14 NAME= specifier in the INQUIRE statement**

- 2 The *scalar-default-char-variable* in the NAME= specifier is assigned the value of the name of the file if
3 the file has a name; otherwise, it becomes undefined.

NOTE 9.65

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. However, the value returned shall be suitable for use as the value of the *file-name-expr* in the FILE= specifier in an OPEN statement.

The processor may return a file name qualified by a user identification, device, directory, or other relevant information.

- 4 The case of the characters assigned to *scalar-default-char-variable* is processor dependent.

5 **9.9.2.15 NAMED= specifier in the INQUIRE statement**

- 6 The *scalar-default-logical-variable* in the NAMED= specifier is assigned the value true if the file has a
7 name; otherwise, it is assigned the value false.

8 **9.9.2.16 NEXTREC= specifier in the INQUIRE statement**

- 9 The *scalar-int-variable* in the NEXTREC= specifier is assigned the value $n + 1$, where n is the record
10 number of the last record read from or written to the connection for direct access by the executing
11 image. If there is a connection but no records have been read or written since the connection, the *scalar-*
12 *int-variable* is assigned the value 1. If there is no connection, the connection is not for direct access,
13 or the position is indeterminate because of a previous error condition, the *scalar-int-variable* becomes
14 undefined. If there are pending data transfer operations for the specified unit, the value assigned is
15 computed as if all the pending data transfers had already completed.

16 **9.9.2.17 NUMBER= specifier in the INQUIRE statement**

- 17 The *scalar-int-variable* in the NUMBER= specifier is assigned the value of the external unit number of
18 the unit that is connected to the file. If there is no unit connected to the file, the value -1 is assigned.

19 **9.9.2.18 OPENED= specifier in the INQUIRE statement**

- 20 Execution of an INQUIRE by file statement causes the *scalar-default-logical-variable* in the OPENED=
21 specifier to be assigned the value true if the file specified is connected to a unit; otherwise, false is
22 assigned. Execution of an INQUIRE by unit statement causes the *scalar-default-logical-variable* to be
23 assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

24 **9.9.2.19 PAD= specifier in the INQUIRE statement**

- 25 The *scalar-default-char-variable* in the PAD= specifier is assigned the value YES or NO, corresponding
26 to the pad mode in effect for a connection for formatted input/output. If there is no connection or if
27 the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value
28 UNDEFINED.

29 **9.9.2.20 PENDING= specifier in the INQUIRE statement**

- 30 The PENDING= specifier is used to determine whether previously pending asynchronous data transfers
31 are complete. A data transfer operation is previously pending if it is pending at the beginning of
32 execution of the INQUIRE statement.

- 1 If an ID= specifier appears and the specified data transfer operation is complete, then the variable
- 2 specified in the PENDING= specifier is assigned the value false and the INQUIRE statement performs
- 3 the wait operation for the specified data transfer.
- 4 If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit
- 5 are complete, then the variable specified in the PENDING= specifier is assigned the value false and the
- 6 INQUIRE statement performs wait operations for all previously pending data transfers for the specified
- 7 unit.
- 8 In all other cases, the variable specified in the PENDING= specifier is assigned the value true and no
- 9 wait operations are performed; in this case the previously pending data transfers remain pending after
- 10 the execution of the INQUIRE statement.

NOTE 9.66

The processor has considerable flexibility in defining when it considers a transfer to be complete. Any of the following approaches could be used:

- (1) 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.
- (2) The INQUIRE statement could wait for all specified data transfers to complete and then always return false for PENDING=.
- (3) The INQUIRE statement could actually test the state of the specified data transfer operations.

11 9.9.2.21 POS= specifier in the INQUIRE statement

- 12 The *scalar-int-variable* in the POS= specifier is assigned the number of the file storage unit immediately
- 13 following the current position of a file connected for stream access. If the file is positioned at its terminal
- 14 position, the variable is assigned a value one greater than the number of the highest-numbered file storage
- 15 unit in the file. If the file is not connected for stream access or if the position of the file is indeterminate
- 16 because of previous error conditions, the variable becomes undefined.

17 9.9.2.22 POSITION= specifier in the INQUIRE statement

- 18 The *scalar-default-char-variable* in the POSITION= specifier is assigned the value REWIND if the
- 19 connection was opened for positioning at its initial point, APPEND if the connection was opened for
- 20 positioning before its endfile record or at its terminal point, and ASIS if the connection was opened
- 21 without changing its position. If there is no connection or if the file is connected for direct access, the
- 22 *scalar-default-char-variable* is assigned the value UNDEFINED. If the file has been repositioned since
- 23 the connection, the *scalar-default-char-variable* is assigned a processor-dependent value, which shall not
- 24 be REWIND unless the file is positioned at its initial point and shall not be APPEND unless the file is
- 25 positioned so that its endfile record is the next record or at its terminal point if it has no endfile record.

26 9.9.2.23 READ= specifier in the INQUIRE statement

- 27 The *scalar-default-char-variable* in the READ= specifier is assigned the value YES if READ is included
- 28 in the set of allowed actions for the file, NO if READ is not included in the set of allowed actions for
- 29 the file, and UNKNOWN if the processor is unable to determine whether READ is included in the set
- 30 of allowed actions for the file.

31 9.9.2.24 READWRITE= specifier in the INQUIRE statement

1 The *scalar-default-char-variable* in the READWRITE= specifier is assigned the value YES if READ-
2 WRITE is included in the set of allowed actions for the file, NO if READWRITE is not included in
3 the set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether
4 READWRITE is included in the set of allowed actions for the file.

5 **9.9.2.25 RECL= specifier in the INQUIRE statement**

6 The *scalar-int-variable* in the RECL= specifier is assigned the value of the record length of a connection
7 for direct access, or the value of the maximum record length of a connection for sequential access. If
8 the connection is for formatted input/output, the length is the number of characters for all records that
9 contain only characters of type default character. If the connection is for unformatted input/output,
10 the length is measured in file storage units. If there is no connection, or if the connection is for stream
11 access, the *scalar-int-variable* becomes undefined.

12 **9.9.2.26 ROUND= specifier in the INQUIRE statement**

13 The *scalar-default-char-variable* in the ROUND= specifier is assigned the value UP, DOWN, ZERO,
14 NEAREST, COMPATIBLE, or PROCESSOR_DEFINED, corresponding to the I/O rounding mode in
15 effect for a connection for formatted input/output. If there is no connection or if the connection is not
16 for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED. The
17 processor shall return the value PROCESSOR_DEFINED only if the I/O rounding mode currently in
18 effect behaves differently than the UP, DOWN, ZERO, NEAREST, and COMPATIBLE modes.

19 **9.9.2.27 SEQUENTIAL= specifier in the INQUIRE statement**

20 The *scalar-default-char-variable* in the SEQUENTIAL= specifier is assigned the value YES if SEQUEN-
21 TIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included
22 in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine
23 whether SEQUENTIAL is included in the set of allowed access methods for the file.

24 **9.9.2.28 SIGN= specifier in the INQUIRE statement**

25 The *scalar-default-char-variable* in the SIGN= specifier is assigned the value PLUS, SUPPRESS, or
26 PROCESSOR_DEFINED, corresponding to the sign mode in effect for a connection for formatted in-
27 put/output. If there is no connection, or if the connection is not for formatted input/output, the
28 *scalar-default-char-variable* is assigned the value UNDEFINED.

29 **9.9.2.29 SIZE= specifier in the INQUIRE statement**

30 The *scalar-int-variable* in the SIZE= specifier is assigned the size of the file in file storage units. If the
31 file size cannot be determined, the variable is assigned the value -1.

32 For a file that may be connected for stream access, the file size is the number of the highest-numbered
33 file storage unit in the file.

34 For a file that may be connected for sequential or direct access, the file size may be different from the
35 number of storage units implied by the data in the records; the exact relationship is processor-dependent.

36 **9.9.2.30 STREAM= specifier in the INQUIRE statement**

37 The *scalar-default-char-variable* in the STREAM= specifier is assigned the value YES if STREAM is
38 included in the set of allowed access methods for the file, NO if STREAM is not included in the set of
39 allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether
40 STREAM is included in the set of allowed access methods for the file.

1 9.9.2.31 TEAM= specifier in the INQUIRE statement

- 2 The *image-team* in the TEAM= specifier is assigned the value of the connect team if the file or unit is
3 connected; otherwise it is assigned a value that identifies an empty image set.

NOTE 9.67

The indices of the images in a team may be obtained by using the intrinsic function TEAM_IMAGES (13.7.172).

4 9.9.2.32 UNFORMATTED= specifier in the INQUIRE statement

- 5 The *scalar-default-char-variable* in the UNFORMATTED= specifier is assigned the value YES if UN-
6 FORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included
7 in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether
8 UNFORMATTED is included in the set of allowed forms for the file.

9 9.9.2.33 WRITE= specifier in the INQUIRE statement

- 10 The *scalar-default-char-variable* in the WRITE= specifier is assigned the value YES if WRITE is included
11 in the set of allowed actions for the file, NO if WRITE is not included in the set of allowed actions for
12 the file, and UNKNOWN if the processor is unable to determine whether WRITE is included in the set
13 of allowed actions for the file.

14 9.9.3 Inquire by output list

- 15 The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent number of file
16 storage units that would be required to store the data of the output list in an unformatted file. The
17 value shall be suitable as a RECL= specifier in an OPEN statement that connects a file for unformatted
18 direct access when there are input/output statements with the same input/output list.

- 19 The output list in an INQUIRE statement shall not contain any derived-type list items that require
20 a user-defined derived-type input/output procedure as described in subclause 9.5.3. If a derived-type
21 list item appears in the output list, the value returned for the IOLENGTH= specifier assumes that no
22 user-defined derived-type input/output procedure will be invoked.

23 9.10 Error, end-of-record, and end-of-file conditions

- 24 The set of input/output error conditions is processor dependent.

- 25 An **end-of-record condition** occurs when a nonadvancing input statement attempts to transfer data
26 from a position beyond the end of the current record, unless the file is a stream file and the current
27 record is at the end of the file (an end-of-file condition occurs instead).

- 28 An **end-of-file condition** occurs when

- 29 (1) an endfile record is encountered during the reading of a file connected for sequential access,
- 30 (2) an attempt is made to read a record beyond the end of an internal file, or
- 31 (3) an attempt is made to read beyond the end of a stream file.

- 32 An end-of-file condition may occur at the beginning of execution of an input statement. An end-of-file
33 condition also may occur during execution of a formatted input statement when more than one record
34 is required by the interaction of the input list and the format. An end-of-file condition also may occur
35 during execution of a stream input statement.

1 9.10.1 Error conditions and the ERR= specifier

2 If an error condition occurs during execution of an input/output statement, the position of the file
3 becomes indeterminate.

4 If an error condition occurs during execution of an input/output statement that contains neither an
5 ERR= nor IOSTAT= specifier, execution of the program is terminated. If an error condition occurs
6 during execution of an input/output statement that contains either an ERR= specifier or an IOSTAT=
7 specifier then

- 8 (1) processing of the input/output list, if any, terminates,
- 9 (2) if the statement is a data transfer statement or the error occurs during a wait operation, all
10 *do-variables* in the statement that initiated the transfer become undefined,
- 11 (3) if an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes
12 defined as specified in 9.10.4,
- 13 (4) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
- 14 (5) if the statement is a READ statement and it contains a SIZE= specifier, the *scalar-int-*
15 *variable* in the SIZE= specifier becomes defined as specified in 9.5.2.15,
- 16 (6) if the statement is a READ statement or the error condition occurs in a wait operation for
17 a transfer initiated by a READ statement, all input items or namelist group objects in the
18 statement that initiated the transfer become undefined, and
- 19 (7) if an ERR= specifier appears, a branch to the statement labeled by the *label* in the ERR=
20 specifier occurs.

21 9.10.2 End-of-file condition and the END= specifier

22 If an end-of-file condition occurs during execution of an input/output statement that contains neither
23 an END= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-file
24 condition occurs during execution of an input/output statement that contains either an END= specifier
25 or an IOSTAT= specifier, and an error condition does not occur then

- 26 (1) processing of the input list, if any, terminates,
- 27 (2) if the statement is a data transfer statement or the error occurs during a wait operation, all
28 *do-variables* in the statement that initiated the transfer become undefined,
- 29 (3) if the statement is a READ statement or the end-of-file condition occurs in a wait operation
30 for a transfer initiated by a READ statement, all input list items or namelist group objects
31 in the statement that initiated the transfer become undefined,
- 32 (4) if the file specified in the input statement is an external record file, it is positioned after the
33 endfile record,
- 34 (5) if an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes
35 defined as specified in 9.10.4,
- 36 (6) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
37 and
- 38 (7) if an END= specifier appears, a branch to the statement labeled by the *label* in the END=
39 specifier occurs.

40 9.10.3 End-of-record condition and the EOR= specifier

41 If an end-of-record condition occurs during execution of an input/output statement that contains neither
42 an EOR= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-
43 record condition occurs during execution of an input/output statement that contains either an EOR=
44 specifier or an IOSTAT= specifier, and an error condition does not occur then

- 45 (1) If the pad mode has the value

- 1 (a) YES, the record is padded with blanks to satisfy the effective list item (9.5.4.4.2)
- 2 and corresponding data edit descriptors that require more characters than the record
- 3 contains,
- 4 (b) NO, the input list item becomes undefined,
- 5 (2) processing of the input list, if any, terminates,
- 6 (3) if the statement is a data transfer statement or the end-of-record condition occurs dur-
- 7 ing a wait operation, all *do-variables* in the statement that initiated the transfer become
- 8 undefined,
- 9 (4) the file specified in the input statement is positioned after the current record,
- 10 (5) if an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes
- 11 defined as specified in 9.10.4,
- 12 (6) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
- 13 (7) if a SIZE= specifier appears, the *scalar-int-variable* in the SIZE= specifier becomes defined
- 14 as specified in (9.5.2.15), and
- 15 (8) if an EOR= specifier appears, a branch to the statement labeled by the *label* in the EOR=
- 16 specifier occurs.

17 9.10.4 IOSTAT= specifier

18 Execution of an input/output statement containing the IOSTAT= specifier causes the *scalar-int-variable*
 19 in the IOSTAT= specifier to become defined with

- 20 (1) a zero value if neither an error condition, an end-of-file condition, nor an end-of-record
- 21 condition occurs,
- 22 (2) the processor-dependent positive integer value of the constant IOSTAT_INQUIRE_INTERNAL-
- 23 UNIT from the ISO_FORTRAN_ENV intrinsic module (13.8.2) if a unit number in an
- 24 INQUIRE statement identifies an internal file,
- 25 (3) a processor-dependent positive integer value different from IOSTAT_INQUIRE_INTERNAL-
- 26 _UNIT if any other error condition occurs,
- 27 (4) the processor-dependent negative integer value of the constant IOSTAT_END (13.8.2.10) if
- 28 an end-of-file condition occurs and no error condition occurs, or
- 29 (5) the processor-dependent negative integer value of the constant IOSTAT_EOR (13.8.2.11) if
- 30 an end-of-record condition occurs and no error condition or end-of-file condition occurs.

NOTE 9.68

An end-of-file condition may occur only for sequential or stream input and an end-of-record condition may occur only for nonadvancing input.

Consider the 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

```

31 9.10.5 IOMSG= specifier

- 1 If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement,
 2 the processor shall assign an explanatory message to *iomsg-variable*. If no such condition occurs, the
 3 processor shall not change the value of *iomsg-variable*.

4 **9.11 Restrictions on input/output statements**

- 5 If a unit, or a file connected to a unit, does not have all of the properties required for the execution of
 6 certain input/output statements, those statements shall not refer to the unit.

- 7 An input/output statement that is executed while another input/output statement is being executed is
 8 called a **recursive input/output statement**.

- 9 A recursive input/output statement shall not identify an external unit that is identified by another
 10 input/output statement being executed except that a child data transfer statement may identify its
 11 parent data transfer statement external unit.

- 12 An input/output statement shall not cause the value of any established format specification to be
 13 modified.

- 14 A recursive input/output statement shall not modify the value of any internal unit except that a recursive
 15 WRITE statement may modify the internal unit identified by that recursive WRITE statement.

- 16 The value of a specifier in an input/output statement shall not depend on any *input-item*, *io-implied-*
 17 *do do-variable*, or on the definition or evaluation of any other specifier in the *io-control-spec-list* or
 18 *inquire-spec-list* in that statement.

- 19 The value of any subscript or substring bound of a variable that appears in a specifier in an input/output
 20 statement shall not depend on any *input-item*, *io-implied-do do-variable*, or on the definition or evalua-
 21 tion of any other specifier in the *io-control-spec-list* or *inquire-spec-list* in that statement.

- 22 In a data transfer statement, the variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier, if
 23 any, shall not be associated with any entity in the data transfer input/output list (9.5.3) or *namelist-*
 24 *group-object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.

- 25 In a data transfer statement, if a variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier is an
 26 array element reference, its subscript values shall not be affected by the data transfer, the *io-implied-do*
 27 processing, or the definition or evaluation of any other specifier in the *io-control-spec-list*.

- 28 A variable that may become defined or undefined as a result of its use in a specifier in an INQUIRE
 29 statement, or any associated entity, shall not appear in another specifier in the same INQUIRE statement.

- 30 A STOP statement shall not be executed during execution of an input/output statement.

NOTE 9.69

Restrictions on the evaluation of expressions (7.1.4) prohibit certain side effects.

10 Input/output editing

10.1 Format specifications

A format used in conjunction with an input/output statement provides information that directs the editing between the internal representation of data and the characters of a sequence of formatted records.

A R914 (9.5.2.2) in an input/output statement may refer to a FORMAT statement or to a character expression that contains a format specification. A format specification provides explicit editing information. The R914 alternatively may be an asterisk (*), which indicates list-directed formatting (10.10). Namelist formatting (10.11) may be indicated by specifying a *namelist-group-name* instead of a *format*.

10.2 Explicit format specification methods

10.2.1 FORMAT statement

R1001 *format-stmt* is FORMAT *format-specification*

R1002 *format-specification* is ([*format-item-list*])
or ([*format-item-list*,] *unlimited-format-item*)

C1001 (R1001) The *format-stmt* shall be labeled.

C1002 (R1002) The comma used to separate *format-items* in a *format-item-list* may be omitted

- (1) between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor (10.8.5), possibly preceded by a repeat specifier,
- (2) before a slash edit descriptor when the optional repeat specification is not present (10.8.2),
- (3) after a slash edit descriptor, or
- (4) before or after a colon edit descriptor (10.8.3)

Blank characters may precede the initial left parenthesis of the format specification. Additional blank characters may appear at any point within the format specification, with no effect on the interpretation of the format specification, except within a character string edit descriptor (10.9).

NOTE 10.1

Examples of FORMAT statements are:

```
5    FORMAT (1PE12.4, I10)
9    FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))
```

10.2.2 Character format specification

A character expression used as a *format* in a formatted input/output statement shall evaluate to a character string whose leading part is a valid format specification.

NOTE 10.2

The format specification begins with a left parenthesis and ends with a right parenthesis.

- 1 All character positions up to and including the final right parenthesis of the format specification shall be
 2 defined at the time the input/output statement is executed, and shall not become redefined or undefined
 3 during the execution of the statement. Character positions, if any, following the right parenthesis that
 4 ends the format specification need not be defined and may contain any character data with no effect on
 5 the interpretation of the format specification.
- 6 If the *format* is a character array, it is treated as if all of the elements of the array were specified in array
 7 element order and were concatenated. However, if a *format* is a character array element, the format
 8 specification shall be entirely within that array element.

NOTE 10.3

If a character constant is used as a *format* in an input/output statement, care shall be taken that the value of the character constant is a valid format specification. In particular, if a format specification delimited by apostrophes contains a character constant edit descriptor delimited with apostrophes, two apostrophes shall be written to delimit the edit descriptor and four apostrophes shall be written for each apostrophe that occurs within the edit descriptor. For example, the text:

```
2 ISN'T 3
```

may 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.

9 10.3 Form of a format item list

10 10.3.1 Syntax

11 R1003 <i>format-item</i>	is [<i>r</i>] <i>data-edit-desc</i>
12	or <i>control-edit-desc</i>
13	or <i>char-string-edit-desc</i>
14	or [<i>r</i>] (<i>format-item-list</i>)
15 R1004 <i>unlimited-format-item</i>	is * (<i>format-item-list</i>)
16 R1005 <i>r</i>	is <i>int-literal-constant</i>
17 C1003 (R1005) <i>r</i> shall be positive.	
18 C1004 (R1005) <i>r</i> shall not have a kind parameter specified for it.	
19	The integer literal constant <i>r</i> is called a repeat specification .

20 10.3.2 Edit descriptors

- 21 An edit descriptor is a **data edit descriptor**, a **control edit descriptor**, or a **character string**
 22 **edit descriptor**.

1 R1006 *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** G *w* [. *d* [E *e*]]
10 **or** L *w*
11 **or** A [*w*]
12 **or** D *w* . *d*
13 **or** DT [*char-literal-constant*] [(*v-list*)]

14 R1007 *w* **is** *int-literal-constant*

15 R1008 *m* **is** *int-literal-constant*

16 R1009 *d* **is** *int-literal-constant*

17 R1010 *e* **is** *int-literal-constant*

18 R1011 *v* **is** *signed-int-literal-constant*

19 C1005 (R1010) *e* shall be positive.

20 C1006 (R1007) *w* shall be zero or positive for the I, B, O, Z, F, and G edit descriptors. *w* shall be
21 positive for all other edit descriptors.

22 C1007 (R1006) For the G edit descriptor, *d* shall be specified if and only if *w* is not zero.

23 C1008 (R1006) *w*, *m*, *d*, *e*, and *v* shall not have kind parameters specified for them.

24 C1009 (R1006) The *char-literal-constant* in the DT edit descriptor shall not have a kind parameter
25 specified for it.

26 I, B, O, Z, F, E, EN, ES, G, L, A, D, and DT indicate the manner of editing.

27 R1012 *control-edit-desc* **is** *position-edit-desc*
28 **or** [*r*] /
29 **or** :
30 **or** *sign-edit-desc*
31 **or** *k* P
32 **or** *blank-interp-edit-desc*
33 **or** *round-edit-desc*
34 **or** *decimal-edit-desc*

35 R1013 *k* **is** *signed-int-literal-constant*

36 C1010 (R1013) *k* shall not have a kind parameter specified for it.

37 In *k* P, *k* is called the **scale factor**.

38 R1014 *position-edit-desc* **is** T *n*
39 **or** TL *n*
40 **or** TR *n*
41 **or** *n* X

1 R1015 *n* **is** *int-literal-constant*

2 C1011 (R1015) *n* shall be positive.

3 C1012 (R1015) *n* shall not have a kind parameter specified for it.

4 R1016 *sign-edit-desc* **is** SS
 5 **or** SP
 6 **or** S

7 R1017 *blank-interp-edit-desc* **is** BN
 8 **or** BZ

9 R1018 *round-edit-desc* **is** RU
 10 **or** RD
 11 **or** RZ
 12 **or** RN
 13 **or** RC
 14 **or** RP

15 R1019 *decimal-edit-desc* **is** DC
 16 **or** DP

17 T, TL, TR, X, slash, colon, SS, SP, S, P, BN, BZ, RU, RD, RZ, RN, RC, RP, DC, and DP indicate the
 18 manner of editing.

19 R1020 *char-string-edit-desc* **is** *char-literal-constant*

20 C1013 (R1020) The *char-literal-constant* shall not have a kind parameter specified for it.

21 Each *rep-char* in a character string edit descriptor shall be one of the characters capable of representation
 22 by the processor.

23 The character string edit descriptors provide constant data to be output, and are not valid for input.

24 The edit descriptors are without regard to case except for the characters in the character constants.

25 10.3.3 Fields

26 A **field** is a part of a record that is read on input or written on output when format control encounters
 27 a data edit descriptor or a character string edit descriptor. The **field width** is the size in characters of
 28 the field.

29 10.4 Interaction between input/output list and format

30 The start of formatted data transfer using a format specification initiates **format control** (9.5.4.4.2).
 31 Each action of format control depends on information jointly provided by the next edit descriptor in the
 32 format specification and the next effective item in the input/output list, if one exists.

33 If an input/output list specifies at least one effective list item, at least one data edit descriptor shall
 34 exist in the format specification.

NOTE 10.4

An empty format specification of the form () may be used only if the input/output list has no effective list items (9.5.4.4). Zero length character items are effective list items, but zero sized arrays and implied DO lists with an iteration count of zero are not.

- 1 A format specification is interpreted from left to right. The exceptions are format items preceded by a
- 2 repeat specification *r*, and format reversion (described below).
- 3 A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the
- 4 format item but without the repeat specification and separated by commas.

NOTE 10.5

An omitted repeat specification is treated in the same way as a repeat specification whose value is one.

- 5 To each data edit descriptor interpreted in a format specification, there corresponds one effective item
- 6 specified by the input/output list (9.5.3), except that an input/output list item of type complex requires
- 7 the interpretation of two F, E, EN, ES, D, or G edit descriptors. For each control edit descriptor or
- 8 character edit descriptor, there is no corresponding item specified by the input/output list, and format
- 9 control communicates information directly with the record.
- 10 Whenever format control encounters a data edit descriptor in a format specification, it determines
- 11 whether there is a corresponding effective item specified by the input/output list. If there is such an
- 12 item, it transmits appropriately edited information between the item and the record, and then format
- 13 control proceeds. If there is no such item, format control terminates.
- 14 If format control encounters a colon edit descriptor in a format specification and another effective in-
- 15 put/output list item is not specified, format control terminates.
- 16 If format control encounters the rightmost parenthesis of a complete format specification and another
- 17 effective input/output list item is not specified, format control terminates. However, if another effective
- 18 input/output list item is specified, format control then reverts to the beginning of the format item
- 19 terminated by the last preceding right parenthesis that is not part of a DT edit descriptor. If there
- 20 is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format
- 21 specification. If any reversion occurs, the reused portion of the format specification shall contain at
- 22 least one data edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat
- 23 specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on
- 24 the changeable modes (9.4.1). If format control reverts to a parenthesis that is not the beginning of an
- 25 *unlimited-format-item*, the file is positioned in a manner identical to the way it is positioned when a
- 26 slash edit descriptor is processed (10.8.2).

NOTE 10.6

Example: The format specification:

```
10 FORMAT (1X, 2(F10.3, I5))
```

with an output list of

```
WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6
```

produces the same output as the format specification:

```
10 FORMAT (1X, F10.3, I5, F10.3, I5/F10.3, I5, F10.3, I5)
```

NOTE 10.7

The effect of an *unlimited-format-item* is as if its enclosed list were preceded by a very large repeat count. There is no file positioning implied by *unlimited-format-item* reversion. This may be used to write what is commonly called a comma separated value record.

NOTE 10.7 (cont.)

For example,

```
WRITE( 10, '( "IARRAY =", *( IO, :, ",") )' ) IARRAY
```

produces a single record with a header and a comma separated list of integer values.

10.5 Positioning by format control

After each data edit descriptor or character string edit descriptor is processed, the file is positioned after the last character read or written in the current record.

After each T, TL, TR, or X edit descriptor is processed, the file is positioned as described in 10.8.1.

After each slash edit descriptor is processed, the file is positioned as described in 10.8.2.

During formatted stream output, processing of an A edit descriptor can cause file positioning to occur (10.7.4).

If format control reverts as described in 10.4, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (10.8.2).

During a read operation, any unprocessed characters of the current record are skipped whenever the next record is read.

10.6 Decimal symbol

The **decimal symbol** is the character that separates the whole and fractional parts in the decimal representation of a real number in an internal or external file. When the decimal edit mode is POINT, the decimal symbol is a decimal point. When the decimal edit mode is COMMA, the decimal symbol is a comma.

10.7 Data edit descriptors

10.7.1 General

Data edit descriptors cause the conversion of data to or from its internal representation; during formatted stream output, the A data edit descriptor may also cause file positioning. On input, the specified variable becomes defined unless an error condition, an end-of-file condition, or an end-of-record condition occurs. On output, the specified expression is evaluated.

During input from a Unicode file,

- (1) characters in the record that correspond to an ASCII character variable shall have a position in the ISO 10646 character type collating sequence of 127 or less, and
- (2) characters in the record that correspond to a default character variable shall be representable in the default character type.

During input from a non-Unicode file,

- (1) characters in the record that correspond to a character variable shall have the kind of the character variable, and
- (2) characters in the record that correspond to a numeric logical, or bits variable shall be of default character type.

1 During output to a Unicode file, all characters transmitted to the record are of ISO 10646 character
 2 type. If a character input/output list item or character string edit descriptor contains a character that
 3 is not representable in the ISO 10646 character type, the result is processor-dependent.

4 During output to a non-Unicode file, characters transmitted to the record as a result of processing a
 5 character string edit descriptor or as a result of evaluating a numeric, logical, bits, or default character
 6 data entity, are of type default character.

7 **10.7.2 Numeric and bits editing**

8 **10.7.2.1 General rules**

9 The I, B, O, Z, F, E, EN, ES, D, and G edit descriptors may be used to specify the input/output of
 10 integer, real, complex, and bits data. The following general rules apply.

- 11 (1) On input, leading blanks are not significant. When the input field is not an IEEE excep-
 12 tional specification (10.7.2.3.2), the interpretation of blanks, other than leading blanks, is
 13 determined by the blank interpretation mode (10.8.6). Plus signs may be omitted. A field
 14 containing only blanks is considered to be zero.
- 15 (2) On input, with F, E, EN, ES, D, and G editing, a decimal symbol appearing in the input
 16 field overrides the portion of an edit descriptor that specifies the decimal symbol location.
 17 The input field may have more digits than the processor uses to approximate the value of
 18 the datum.
- 19 (3) On output with I, F, E, EN, ES, D, and G editing, the representation of a positive or zero
 20 internal value in the field may be prefixed with a plus sign, as controlled by the S, SP, and
 21 SS edit descriptors or the processor. The representation of a negative internal value in the
 22 field shall be prefixed with a minus sign.
- 23 (4) On output, the representation is right justified in the field. If the number of characters
 24 produced by the editing is smaller than the field width, leading blanks are inserted in the
 25 field.
- 26 (5) On output, if the number of characters produced exceeds the field width or if an exponent
 27 exceeds its specified length using the *Ew.d Ee*, *ENw.d Ee*, *ESw.d Ee*, or *Gw.d Ee* edit
 28 descriptor, the processor shall fill the entire field of width *w* with asterisks. However,
 29 the processor shall not produce asterisks if the field width is not exceeded when optional
 30 characters are omitted.

NOTE 10.8

When an SP edit descriptor is in effect, a plus sign is not optional.

- 31 (6) On output, with I, B, O, Z, F, and G editing, the specified value of the field width *w* may
 32 be zero. In such cases, the processor selects the smallest positive actual field width that
 33 does not result in a field filled with asterisks. The specified value of *w* shall not be zero on
 34 input.

35 **10.7.2.2 Integer editing**

36 The *Iw* and *Iw.m*, edit descriptors indicate that the field to be edited occupies *w* positions, except when
 37 *w* is zero. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero. The
 38 specified input/output list item shall be of type integer or bits. The G, B, O, and Z edit descriptor also
 39 may be used to edit integer data (10.7.5.2.1, 10.7.2.4).

40 If the input list item is of type bits, the integer value specified by the input field is converted to type
 41 bits according to the model in 13.3. On input, *m* has no effect.

- 1 In the input field for the I edit descriptor, the character string shall be a *signed-digit-string* (R409) if
 2 the list item is of type integer and a *digit-string* (R410) if it is of type bits, except for the interpretation
 3 of blanks.
- 4 The output field for the Iw edit descriptor consists of zero or more leading blanks followed by a minus
 5 sign if the internal value is negative, or an optional plus sign otherwise, followed by the magnitude of
 6 the internal value as a *digit-string* without leading zeros.

NOTE 10.9

A <i>digit-string</i> always consists of at least one digit.

- 7 The output field for the Iw.m edit descriptor is the same as for the Iw edit descriptor, except that the
 8 *digit-string* consists of at least *m* digits. If necessary, sufficient leading zeros are included to achieve the
 9 minimum of *m* digits. The value of *m* shall not exceed the value of *w*, except when *w* is zero. If *m* is
 10 zero and the internal value is zero, the output field consists of only blank characters, regardless of the
 11 sign control in effect. When *m* and *w* are both zero, and the internal value is zero, one blank character
 12 is produced.

10.7.2.3 Real and complex editing**10.7.2.3.1 General**

- 15 The F, E, EN, ES, and D edit descriptors specify the editing of real and complex data. An input/output
 16 list item corresponding to an F, E, EN, ES, or D edit descriptor shall be real or complex. The G, B, O,
 17 and Z edit descriptors also may be used to edit real and complex data (10.7.5.2.2).

10.7.2.3.2 F editing

- 19 The Fw.d edit descriptor indicates that the field occupies *w* positions, the fractional part of which
 20 consists of *d* digits. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero.

- 21 A lower-case letter is equivalent to the corresponding upper-case letter in an IEEE exceptional specifi-
 22 cation or the exponent in a numeric input field.

- 23 The input field is either an IEEE exceptional specification or consists of an optional sign, followed by a
 24 string of one or more digits optionally containing a decimal symbol, including any blanks interpreted as
 25 zeros. The *d* has no effect on input if the input field contains a decimal symbol. If the decimal symbol
 26 is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted
 27 as the fractional part of the value represented. The string of digits may contain more digits than a
 28 processor uses to approximate the value. The basic form may be followed by an exponent of one of the
 29 following forms:

- 30 • a *sign* followed by a *digit-string*;
- 31 • the letter E followed by zero or more blanks, followed by a *signed-digit-string*;
- 32 • the letter D followed by zero or more blanks, followed by a *signed-digit-string*.

- 33 An exponent containing a D is processed identically to an exponent containing an E.

NOTE 10.10

If the input field does not contain an exponent, the effect is as if the basic form were followed by an exponent with a value of $-k$, where <i>k</i> is the established scale factor (10.8.5).

- 1 An input field that is an IEEE exceptional specification consists of optional blanks, followed by either
- 2 • an optional sign, followed by the string 'INF' or the string 'INFINITY', or
 - 3 • an optional sign, followed by the string 'NaN', optionally followed by zero or more alphanumeric
 - 4 characters enclosed in parentheses,

5 optionally followed by blanks.

6 The value specified by form (1) is an IEEE infinity; this form shall not be used if the processor does
 7 not support IEEE infinities for the input variable. The value specified by form (2) is an IEEE NaN;
 8 this form shall not be used if the processor does not support IEEE NaNs for the input variable. The
 9 NaN value is a quiet NaN if the only nonblank characters in the field are 'NaN' or 'NaN()'; otherwise,
 10 the NaN value is processor-dependent. The interpretation of a sign in a NaN input field is processor
 11 dependent.

12 For an internal value that is an IEEE infinity, the output field consists of blanks, if necessary, followed
 13 by a minus sign for negative infinity or an optional plus sign otherwise, followed by the letters 'Inf' or
 14 'Infinity', right justified within the field. If w is less than 3, the field is filled with asterisks; otherwise,
 15 if w is less than 8, 'Inf' is produced.

16 For an internal value that is an IEEE NaN, the output field consists of blanks, if necessary, followed by
 17 the letters 'NaN' and optionally followed by one to $w - 5$ alphanumeric processor-dependent characters
 18 enclosed in parentheses, right justified within the field. If w is less than 3, the field is filled with asterisks.

NOTE 10.11

The processor-dependent characters following 'NaN' may convey additional information about that particular NaN.

19 For an internal value that is neither an IEEE infinity nor a NaN, the output field consists of blanks, if
 20 necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise,
 21 followed by a string of digits that contains a decimal symbol and represents the magnitude of the internal
 22 value, as modified by the established scale factor and rounded (10.7.2.3.7) to d fractional digits. Leading
 23 zeros are not permitted except for an optional zero immediately to the left of the decimal symbol if the
 24 magnitude of the value in the output field is less than one. The optional zero shall appear if there would
 25 otherwise be no digits in the output field.

26 10.7.2.3.3 E and D editing

27 The $Ew.d$, $Dw.d$, and $Ew.d Ee$ edit descriptors indicate that the external field occupies w positions, the
 28 fractional part of which consists of d digits, unless a scale factor greater than one is in effect, and the
 29 exponent part consists of e digits. The e has no effect on input.

30 The form and interpretation of the input field is the same as for $Fw.d$ editing (10.7.2.3.2).

31 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for
 32 $Fw.d$.

33 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field for a scale
 34 factor of zero is

35 $[\pm] [0].x_1x_2 \dots x_d exp$

36 where:

- 37 • $\pm$ signifies a plus sign or a minus sign;

- 1 • . signifies a decimal symbol (10.6);
- 2 • $x_1x_2 \dots x_d$ are the d most significant digits of the internal value after rounding (10.7.2.3.7);
- 3 • exp is a decimal exponent having one of the forms specified in table 10.1.

Table 10.1: **E and D exponent forms**

Edit Descriptor	Absolute Value of Exponent	Form of Exponent ¹
$Ew.d$	$ exp \leq 99$	$E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < exp \leq 999$	$\pm z_1z_2z_3$
$Ew.d Ee$	$ exp \leq 10^e - 1$	$E\pm z_1z_2 \dots z_e$
$Dw.d$	$ exp \leq 99$	$D\pm z_1z_2$ or $E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 < exp \leq 999$	$\pm z_1z_2z_3$
(1) where each z is a digit.		

- 4 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit
 5 descriptor forms $Ew.d$ and $Dw.d$ shall not be used if $|exp| > 999$.
- 6 The scale factor k controls the decimal normalization (10.3.2, 10.8.5). If $-d < k \leq 0$, the output field
 7 contains exactly $|k|$ leading zeros and $d - |k|$ significant digits after the decimal symbol. If $0 < k < d + 2$,
 8 the output field contains exactly k significant digits to the left of the decimal symbol and $d - k + 1$
 9 significant digits to the right of the decimal symbol. Other values of k are not permitted.

10 10.7.2.3.4 EN editing

- 11 The EN edit descriptor produces an output field in the form of a real number in engineering notation
 12 such that the decimal exponent is divisible by three and the absolute value of the significand (R414) is
 13 greater than or equal to 1 and less than 1000, except when the output value is zero. The scale factor
 14 has no effect on output.

- 15 The forms of the edit descriptor are $ENw.d$ and $ENw.d Ee$ indicating that the external field occupies w
 16 positions, the fractional part of which consists of d digits and the exponent part consists of e digits.

- 17 The form and interpretation of the input field is the same as for $Fw.d$ editing (10.7.2.3.2).

- 18 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for
 19 $Fw.d$.

- 20 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

21 $[\pm] yyy . x_1x_2 \dots x_d exp$

22 where:

- 23 • $\pm$ signifies a plus sign or a minus sign;
- 24 • yyy are the 1 to 3 decimal digits representative of the most significant digits of the internal value
 25 after rounding (10.7.2.3.7);
- 26 • yyy is an integer such that $1 \leq yyy < 1000$ or, if the output value is zero, $yyy = 0$;
- 27 • . signifies a decimal symbol (10.6);
- 28 • $x_1x_2 \dots x_d$ are the d next most significant digits of the internal value after rounding;

- 1 • exp is a decimal exponent, divisible by three, having one of the forms specified in table 10.2.

Table 10.2: **EN exponent forms**

Edit Descriptor	Absolute Value of Exponent	Form of Exponent ¹
EN $w.d$	$ exp \leq 99$	$E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < exp \leq 999$	$\pm z_1 z_2 z_3$
EN $w.d Ee$	$ exp \leq 10^e - 1$	$E\pm z_1 z_2 \dots z_e$
(1) where each z is a digit.		

- 2 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit
3 descriptor form EN $w.d$ shall not be used if $|exp| > 999$.

NOTE 10.12

Examples:

Internal Value	Output field Using SS, EN12.3
6.421	6.421E+00
-.5	-500.000E-03
.00217	2.170E-03
4721.3	4.721E+03

10.7.2.3.5 ES editing

- 5 The ES edit descriptor produces an output field in the form of a real number in scientific notation such
6 that the absolute value of the significand (R414) is greater than or equal to 1 and less than 10, except
7 when the output value is zero. The scale factor has no effect on output.

- 8 The forms of the edit descriptor are ES $w.d$ and ES $w.d Ee$ indicating that the external field occupies w
9 positions, the fractional part of which consists of d digits and the exponent part consists of e digits.

- 10 The form and interpretation of the input field is the same as for F $w.d$ editing (10.7.2.3.2).

- 11 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for
12 F $w.d$.

- 13 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is

14
$$[\pm] y . x_1 x_2 \dots x_d exp$$

- 15 where:

- 16 • $\pm$ signifies a plus sign or a minus sign;
17 • y is a decimal digit representative of the most significant digit of the internal value after rounding
18 (10.7.2.3.7);
19 • $.$ signifies a decimal symbol (10.6);
20 • $x_1 x_2 \dots x_d$ are the d next most significant digits of the internal value after rounding;
21 • exp is a decimal exponent having one of the forms specified in table 10.3.

Table 10.3: **ES exponent forms**

Edit Descriptor	Absolute Value of Exponent	Form of Exponent ¹
<i>ESw.d</i>	$ exp \leq 99$	$E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 < exp \leq 999$	$\pm z_1 z_2 z_3$
<i>ESw.d Ee</i>	$ exp \leq 10^e - 1$	$E\pm z_1 z_2 \dots z_e$
(1) where each z is a digit.		

- 1 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit
2 descriptor form *ESw.d* shall not be used if $|exp| > 999$.

NOTE 10.13

Examples:

Internal Value	Output field Using SS, ES12.3
6.421	6.421E+00
-.5	-5.000E-01
.00217	2.170E-03
4721.3	4.721E+03

3 10.7.2.3.6 Complex editing

- 4 A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex type
5 is specified by two edit descriptors each of which specifies the editing of real data. The first of the edit
6 descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may
7 be different. Control and character string edit descriptors may be processed between the edit descriptor
8 for the real part and the edit descriptor for the imaginary part.

9 10.7.2.3.7 Rounding mode

- 10 The rounding mode can be specified by an OPEN statement (9.4.1), a data transfer input/output
11 statement (9.5.2.13), or an edit descriptor (10.8.7).

- 12 In what follows, the term “decimal value” means the exact decimal number as given by the character
13 string, while the term “internal value” means the number actually stored in the processor. For example,
14 in dealing with the decimal constant 0.1, the decimal value is the mathematical quantity 1/10, which
15 has no exact representation in binary form. Formatted output of real data involves conversion from an
16 internal value to a decimal value; formatted input involves conversion from a decimal value to an internal
17 value.

- 18 When the I/O rounding mode is UP, the value resulting from conversion shall be the smallest repre-
19 sentable value that is greater than or equal to the original value. When the I/O rounding mode is
20 DOWN, the value resulting from conversion shall be the largest representable value that is less than or
21 equal to the original value. When the I/O rounding mode is ZERO, the value resulting from conversion
22 shall be the value closest to the original value and no greater in magnitude than the original value. When
23 the I/O rounding mode is NEAREST, the value resulting from conversion shall be the closer of the two
24 nearest representable values if one is closer than the other. If the two nearest representable values are
25 equidistant from the original value, it is processor dependent which one of them is chosen. When the
26 I/O rounding mode is COMPATIBLE, the value resulting from conversion shall be the closer of the
27 two nearest representable values or the value away from zero if halfway between them. When the I/O
28 rounding mode is PROCESSOR_DEFINED, rounding during conversion shall be a processor-dependent
29 default mode, which may correspond to one of the other modes.

- 30 On processors that support IEEE rounding on conversions, NEAREST shall correspond to round to

- 1 nearest, as specified in the IEEE International Standard.

NOTE 10.14

On processors that support IEEE rounding on conversions, the I/O rounding modes COMPATIBLE and NEAREST will produce the same results except when the datum is halfway between the two representable values. In that case, NEAREST will pick the even value, but COMPATIBLE will pick the value away from zero. The I/O rounding modes UP, DOWN, and ZERO have the same effect as those specified in the IEEE International Standard for round toward $+\infty$, round toward $-\infty$, and round toward 0, respectively.

2 10.7.2.4 Bits editing

- 3 The *Bw*, *Bw.m*, *Ow*, *Ow.m*, *Zw*, and *Zw.m* edit descriptors indicate that the field to be edited occupies
4 *w* positions, except when *w* is zero. When *w* is zero, the processor selects the field width. On input,
5 *w* shall not be zero. The input/output list item shall be of type bits, integer, real, complex, or logical.
6 The G edit descriptor (10.7.5.3) or the I edit descriptor (10.7.2.2) also can be used to edit bits data.

- 7 If the input list item is not of type bits, the input field is edited as if the input list item were of type
8 bits and bits compatible (12.5.2.4) with the actual list item.

- 9 On input, *m* has no effect.

- 10 In the input field for the B, O, and Z edit descriptors the character string shall consist of binary, octal,
11 or hexadecimal digits (as in R426, R427, R428) in the respective input field. The lower-case hexadecimal
12 digits a through f in a hexadecimal input field are equivalent to the corresponding upper-case hexadecimal
13 digits.

- 14 If the output list item, *x*, is not of type bits, it is interpreted as if it were of type bits with the value
15 BITS(*x*).

- 16 The output field for the *Bw*, *Ow*, and *Zw* descriptors consists of zero or more leading blanks followed by
17 the internal value in a form identical to the digits of a binary, octal, or hexadecimal constant, respectively,
18 with the same value and without leading zeros.

NOTE 10.15

A binary, octal, or hexadecimal constant always consists of at least one digit or hexadecimal digit.

- 19 R1021 *hex-digit-string* is *hex-digit* [*hex-digit*] ...

- 20 The output field for the *Bw.m*, *Ow.m*, and *Zw.m* edit descriptor is the same as for the *Bw*, *Ow*, and *Zw*
21 edit descriptor, except that the *digit-string* or *hex-digit-string* consists of at least *m* digits. If necessary,
22 sufficient leading zeros are included to achieve the minimum of *m* digits. The value of *m* shall not exceed
23 the value of *w*, except when *w* is zero. If *m* is zero and the internal value consists of all zero bits, the
24 output field consists of only blank characters. When *m* and *w* are both zero, and the internal value
25 consists of all zero bits, one blank character is produced.

26 10.7.3 Logical editing

- 27 The *Lw* edit descriptor indicates that the field occupies *w* positions. The specified input/output list
28 item shall be of type logical. The G edit descriptor also may be used to edit logical data (10.7.5.4).

- 29 The input field consists of optional blanks, optionally followed by a period, followed by a T for true or
30 F for false. The T or F may be followed by additional characters in the field, which are ignored.

- 31 A lower-case letter is equivalent to the corresponding upper-case letter in a logical input field.

NOTE 10.16

The logical constants `.TRUE.` and `.FALSE.` are acceptable input forms.

- 1 The output field consists of $w - 1$ blanks followed by a T or F, depending on whether the internal value
 2 is true or false, respectively.

3 **10.7.4 Character editing**

- 4 The `A[w]` edit descriptor is used with an input/output list item of type character. The `G` edit descriptor
 5 also may be used to edit character data (10.7.5.5). The kind type parameter of all characters transferred
 6 and converted under control of one `A` or `G` edit descriptor is implied by the kind of the corresponding
 7 list item.

- 8 If a field width w is specified with the `A` edit descriptor, the field consists of w characters. If a field
 9 width w is not specified with the `A` edit descriptor, the number of characters in the field is the length of
 10 the corresponding list item, regardless of the value of the kind type parameter.

- 11 Let len be the length of the input/output list item. If the specified field width w for an `A` edit descriptor
 12 corresponding to an input item is greater than or equal to len , the rightmost len characters will be taken
 13 from the input field. If the specified field width w is less than len , the w characters will appear left
 14 justified with $len - w$ trailing blanks in the internal value.

- 15 If the specified field width w for an `A` edit descriptor corresponding to an output item is greater than
 16 len , the output field will consist of $w - len$ blanks followed by the len characters from the internal value.
 17 If the specified field width w is less than or equal to len , the output field will consist of the leftmost w
 18 characters from the internal value.

NOTE 10.17

For nondefault character types, the blank padding character is processor dependent.

- 19 If the file is connected for stream access, the output may be split across more than one record if it
 20 contains newline characters. A newline character is a nonblank character returned by the intrinsic
 21 function `NEW_LINE`. Beginning with the first character of the output field, each character that is not
 22 a newline is written to the current record in successive positions; each newline character causes file
 23 positioning at that point as if by slash editing (the current record is terminated at that point, a new
 24 empty record is created following the current record, this new record becomes the last and current record
 25 of the file, and the file is positioned at the beginning of this new record).

NOTE 10.18

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.

26 **10.7.5 Generalized editing**

27 **10.7.5.1 Overview**

- 28 The `Gw`, `Gw.d` and `Gw.d Ee` edit descriptors are used with an input/output list item of any intrinsic
 29 type. When w is nonzero, these edit descriptors indicate that the external field occupies w positions;
 30 for real or complex data the fractional part consists of a maximum of d digits and the exponent part
 31 consists of e digits. When these edit descriptors are used to specify the input/output of integer, logical,
 32 bits, or character data, d and e have no effect. When w is zero the processor selects the field width. On
 33 input, w shall not be zero.

1 10.7.5.2 Generalized numeric editing

2 When used to specify the input/output of integer, real, and complex data, the *Gw*, *Gw.d* and *Gw.d Ee*
3 edit descriptors follow the general rules for numeric editing (10.7.2).

NOTE 10.19

The *Gw.d Ee* edit descriptor follows any additional rules for the *Ew.d Ee* edit descriptor.

4 10.7.5.2.1 Generalized integer editing

5 When used to specify the input/output of integer data, the *Gw.d* and *Gw.d Ee* edit descriptors follow
6 the rules for the *Iw* edit descriptor (10.7.2.2), except that *w* shall not be zero. When used to specify the
7 output of integer data, the *G0* edit descriptor follows the rules for the *I0* edit descriptor.

8 10.7.5.2.2 Generalized real and complex editing

9 The form and interpretation of the input field is the same as for *Fw.d* editing (10.7.2.3.2).

10 When used to specify the output of real or complex data, the *G0* edit descriptor follows the rules for
11 the *ESw.d Ee* edit descriptor. Reasonable processor-dependent values of *w*, *d*, and *e* are used with each
12 output value.

13 For an internal value that is an IEEE infinity or NaN, the form of the output field for the *Gw.d* and
14 *Gw.d Ee* edit descriptors is the same as for *Fw.d*.

15 Otherwise, the method of representation in the output field depends on the magnitude of the internal
16 value being edited. Let *N* be the magnitude of the internal value and *r* be the rounding mode value
17 defined in the table below. If $0 < N < 0.1 - r \times 10^{-d-1}$ or $N \geq 10^d - r$, or *N* is identically 0 and
18 *d* is 0, *Gw.d* output editing is the same as *k PEw.d* output editing and *Gw.d Ee* output editing is
19 the same as *k PEw.d Ee* output editing, where *k* is the scale factor (10.8.5) currently in effect. If
20 $0.1 - r \times 10^{-d-1} \leq N < 10^d - r$ or *N* is identically 0 and *d* is not zero, the scale factor has no effect,
21 and the value of *N* determines the editing as follows:

Magnitude of Internal Value	Equivalent Conversion
$N = 0$	$F(w - n).(d - 1), n('b')$
$0.1 - r \times 10^{-d-1} \leq N < 1 - r \times 10^{-d}$	$F(w - n).d, n('b')$
$1 - r \times 10^{-d} \leq N < 10 - r \times 10^{-d+1}$	$F(w - n).(d - 1), n('b')$
$10 - r \times 10^{-d+1} \leq N < 100 - r \times 10^{-d+2}$	$F(w - n).(d - 2), n('b')$
.	.
.	.
.	.
$10^{d-2} - r \times 10^{-2} \leq N < 10^{d-1} - r \times 10^{-1}$	$F(w - n).1, n('b')$
$10^{d-1} - r \times 10^{-1} \leq N < 10^d - r$	$F(w - n).0, n('b')$

22 where *b* is a blank, *n* is 4 for *Gw.d* and *e* + 2 for *Gw.d Ee*, and *r* is defined for each rounding mode as
23 follows:

I/O Rounding Mode	<i>r</i>
COMPATIBLE	0.5
NEAREST	0.5 if the higher value is even −0.5 if the lower value is even
UP	1
DOWN	0

I/O Rounding Mode	<i>r</i>
ZERO	1 if internal value is negative 0 if internal value is positive

- 1 The value of $w - n$ shall be positive

NOTE 10.20

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.

2 **10.7.5.3 Generalized bits editing**

- 3 When used to specify the input/output of bits data, the *Gw.d* and *Gw.d Ee* edit descriptors follow the
4 rules for the *Zw* edit descriptor (10.7.2.4), except that w shall not be zero. When used to specify the
5 output of bits data, the *G0* edit descriptor follows the rules for the *Z0* edit descriptor.

6 **10.7.5.4 Generalized logical editing**

- 7 When used to specify the input/output of logical data, the *Gw.d* and *Gw.d Ee* edit descriptors follow
8 the rules for the *Lw* edit descriptor (10.7.3). When used to specify the output of logical data, the *G0*
9 edit descriptor follows the rules for the *L1* edit descriptor.

10 **10.7.5.5 Generalized character editing**

- 11 When used to specify the input/output of character data, the *Gw.d* and *Gw.d Ee* edit descriptors follow
12 the rules for the *Aw* edit descriptor (10.7.4). When used to specify the output of character data, the *G0*
13 edit descriptor follows the rules for the *A* edit descriptor with no field width.

14 **10.7.6 User-defined derived-type editing**

- 15 The DT edit descriptor allows a user-provided procedure to be used instead of the processor's default
16 input/output formatting for processing a list item of derived type.

- 17 The DT edit descriptor may include a character literal constant. The character value "DT" concatenated
18 with the character literal constant is passed to the user-defined derived-type input/output procedure as
19 the *iotype* argument (9.5.4.7). The v values of the edit descriptor are passed to the user-defined
20 derived-type input/output procedure as the *v_list* array argument.

NOTE 10.21

For the edit descriptor DT'Link List'(10, 4, 2), *iotype* is "DTLink List" and *v_list* is [10, 4, 2].

- 21 If a derived-type variable or value corresponds to a DT edit descriptor, there shall be an accessible
22 interface to a corresponding derived-type input/output procedure for that derived type (9.5.4.7). A DT
23 edit descriptor shall not correspond with a list item that is not of a derived type.

24 **10.8 Control edit descriptors**

- 25 A control edit descriptor does not cause the transfer of data or the conversion of data to or from internal
26 representation, but may affect the conversions performed by subsequent data edit descriptors.

10.8.1 Position editing

The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted to or from the record. If any character skipped by a T, TL, TR, or X edit descriptor is of type nondefault character, and the unit is an internal file of type default character or an external non-Unicode file, the result of that position editing is processor dependent.

The position specified by a T edit descriptor may be in either direction from the current position. On input, this allows portions of a record to be processed more than once, possibly with different editing.

The position specified by an X edit descriptor is forward from the current position. On input, a position beyond the last character of the record may be specified if no characters are transmitted from such positions.

NOTE 10.22

An nX edit descriptor has the same effect as a TRn edit descriptor.

On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and therefore does not by itself affect the length of the record. If characters are transmitted to positions at or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never directly causes a character already placed in the record to be replaced. Such edit descriptors may result in positioning such that subsequent editing causes a replacement.

10.8.1.1 T, TL, and TR editing

The **left tab limit** affects file positioning by the T and TL edit descriptors. Immediately prior to nonchild data transfer, 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.

The Tn edit descriptor indicates that the transmission of the next character to or from a record is to occur at the n th character position of the record, relative to the left tab limit.

The TLn edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position n characters backward from the current position. However, if n is greater than the difference between the current position and the left tab limit, the TLn edit descriptor indicates that the transmission of the next character to or from the record is to occur at the left tab limit.

The TRn edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position n characters forward from the current position.

NOTE 10.23

The n in a Tn , TLn , or TRn edit descriptor shall be specified and shall be greater than zero.

10.8.1.2 X editing

The nX edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position n characters forward from the current position.

NOTE 10.24

The n in an nX edit descriptor shall be specified and shall be greater than zero.

10.8.2 Slash editing

The slash edit descriptor indicates the end of data transfer to or from the current record.

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.

For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number, if there is such a record, and this record becomes the current record.

NOTE 10.25

A record that contains no characters may 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 may be skipped on input.

The repeat specification is optional in the slash edit descriptor. If it is not specified, the default value is one.

10.8.3 Colon editing

The colon edit descriptor terminates format control if there are no more effective items in the input/output list (9.5.3). The colon edit descriptor has no effect if there are more effective items in the input/output list.

10.8.4 SS, SP, and S editing

The SS, SP, and S edit descriptors temporarily change (9.4.1) the sign mode (9.4.5.15, 9.5.2.14) for the connection. The edit descriptors SS, SP, and S set the sign mode corresponding to the SIGN= specifier values SUPPRESS, PLUS, and PROCESSOR_DEFINED, respectively.

The sign mode controls optional plus characters in numeric output fields. When the sign mode is PLUS, the processor shall produce a plus sign in any position that normally contains an optional plus sign. When the sign mode is SUPPRESS, the processor shall not produce a plus sign in such positions. When the sign mode is PROCESSOR_DEFINED, the processor has the option of producing a plus sign or not in such positions, subject to 10.7.2(5).

The SS, SP, and S edit descriptors affect only I, F, E, EN, ES, D, and G editing during the execution of an output statement. The SS, SP, and S edit descriptors have no effect during the execution of an input statement.

10.8.5 P editing

The kP edit descriptor temporarily changes (9.4.1) the scale factor for the connection to k . The scale factor affects the editing of F, E, EN, ES, D, and G edit descriptors for numeric quantities.

The scale factor k affects the appropriate editing in the following manner.

- (1) On input, with F, E, EN, ES, D, and G editing (provided that no exponent exists in the field) and F output editing, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10^k .
- (2) On input, with F, E, EN, ES, D, and G editing, the scale factor has no effect if there is an exponent in the field.
- (3) On output, with E and D editing, the significand (R414) part of the quantity to be produced is multiplied by 10^k and the exponent is reduced by k .
- (4) On output, with G editing, the effect of the scale factor 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.
- (5) On output, with EN and ES editing, the scale factor has no effect.

If UP, DOWN, ZERO, or NEAREST I/O rounding mode is in effect,

- (1) on input, the scale factor is applied to the external decimal value and then this is converted using the current I/O rounding mode, and
- (2) on output, the internal value is converted using the current I/O rounding mode and then the scale factor is applied to the converted decimal value.

10.8.6 BN and BZ editing

The BN and BZ edit descriptors temporarily change (9.4.1) the blank interpretation mode (9.4.5.4, 9.5.2.6) for the connection. The edit descriptors BN and BZ set the blank interpretation mode corresponding to the BLANK= specifier values NULL and ZERO, respectively.

The blank interpretation mode controls the interpretation of nonleading blanks in numeric and bits 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.

The blank interpretation mode affects only numeric editing, bits editing, generalized numeric editing, and generalized bits editing on input. It has no effect on output.

10.8.7 RU, RD, RZ, RN, RC, and RP editing

The round edit descriptors temporarily change (9.4.1) the connection's I/O rounding mode (9.4.5.14, 9.5.2.13, 10.7.2.3.7). The round edit descriptors RU, RD, RZ, RN, RC, and RP set the I/O rounding mode corresponding to the ROUND= specifier values UP, DOWN, ZERO, NEAREST, COMPATIBLE, and PROCESSOR_DEFINED, respectively. The I/O rounding mode affects the conversion of real and complex values in formatted input/output. It affects only D, E, EN, ES, F, and G editing.

10.8.8 DC and DP editing

The decimal edit descriptors temporarily change (9.4.1) the decimal edit mode (9.4.5.5, 9.5.2.7, 10.6) for the connection. The edit descriptors DC and DP set the decimal edit mode corresponding to the DECIMAL= specifier values COMMA and POINT, respectively.

The decimal edit mode controls the representation of the decimal symbol (10.6) during conversion of real and complex values in formatted input/output. The decimal edit mode affects only D, E, EN, ES, F, and G editing. If the mode is COMMA during list-directed input/output, the character used as a value separator is a semicolon in place of a comma.

10.9 Character string edit descriptors

A character string edit descriptor shall not be used on input.

The character string edit descriptor causes characters to be written from the enclosed characters of the edit descriptor itself, including blanks. For a character string edit descriptor, the width of the field is the number of characters between the delimiting characters. Within the field, two consecutive delimiting characters are counted as a single character.

NOTE 10.26

A delimiter for a character string edit descriptor is either an apostrophe or quote.

10.10 List-directed formatting

10.10.1 General

List-directed input/output allows data editing according to the type of the list item instead of by a format specification. It also allows data to be free-field, that is, separated by commas (or semicolons) or blanks.

10.10.2 Values and value separators

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.

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 unlimited character constant and r is an unsigned, nonzero, integer literal constant. Neither c nor r shall have kind type parameters specified. The constant c is interpreted as though it had the same kind type parameter as the corresponding list item. The r^*c form is equivalent to r successive appearances of the constant c , and the r^* form is equivalent to r successive appearances of the null value. Neither of these forms may contain embedded blanks, except where permitted within the constant c .

A **value separator** is

- (1) 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,
- (2) a slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks, or
- (3) one or more contiguous blanks between two nonblank values or following the last nonblank value, where a nonblank value is a constant, an r^*c form, or an r^* form.

NOTE 10.27

Although a slash encountered in an input record is referred to as a separator, it actually causes termination of list-directed and namelist input statements; it does not actually separate two values.

NOTE 10.28

If no list items are specified in a list-directed input/output statement, one input record is skipped or one empty output record is written.

1 10.10.3 List-directed input

2 Input forms acceptable to edit descriptors for a given type are acceptable for list-directed formatting,
 3 except as noted below. The form of the input value shall be acceptable for the type of the next effective
 4 item in the list. Blanks are never used as zeros, and embedded blanks are not permitted in constants,
 5 except within character constants and complex constants as specified below.

6 For the r^*c form of an input value, the constant c is interpreted as an undelimited character constant
 7 if the first list item corresponding to this value is of type default, ASCII, or ISO 10646 character, there
 8 is a nonblank character immediately after r^* , and that character is not an apostrophe or a quotation
 9 mark; otherwise, c is interpreted as a literal constant.

NOTE 10.29

The end of a record has the effect of a blank, except when it appears within a character constant.

10 When the next effective item is of type integer, the value in the input record is interpreted as if an Iw
 11 edit descriptor with a suitable value of w were used.

12 When the next effective item is of type bits, the value in the input record is interpreted as if a Zw edit
 13 descriptor with a suitable value of w were used.

14 When the next effective item is of type real, the input form is that of a numeric input field. A numeric
 15 input field is a field suitable for F editing (10.7.2.3.2) that is assumed to have no fractional digits unless
 16 a decimal symbol appears within the field.

17 When the next effective item is of type complex, the input form consists of a left parenthesis followed by
 18 an ordered pair of numeric input fields separated by a comma (if the decimal edit mode is POINT) or
 19 semicolon (if the decimal edit mode is COMMA), and followed by a right parenthesis. The first numeric
 20 input field is the real part of the complex constant and the second is the imaginary part. Each of the
 21 numeric input fields may be preceded or followed by any number of blanks and ends of records. The end
 22 of a record may occur after the real part or before the imaginary part.

23 When the next effective item is of type logical, the input form shall not include value separators among
 24 the optional characters permitted for L editing.

25 When the next effective item is of type character, the input form consists of a possibly delimited sequence
 26 of zero or more *rep-chars* whose kind type parameter is implied by the kind of the effective list item.
 27 Character sequences may be continued from the end of one record to the beginning of the next record,
 28 but the end of record shall not occur between a doubled apostrophe in an apostrophe-delimited character
 29 sequence, nor between a doubled quote in a quote-delimited character sequence. The end of the record
 30 does not cause a blank or any other character to become part of the character sequence. The character
 31 sequence may be continued on as many records as needed. The characters blank, comma, semicolon,
 32 and slash may appear in default, ASCII, or ISO 10646 character sequences.

33 If the next effective item is of type default, ASCII, or ISO 10646 character and

- 34 (1) the character sequence does not contain value separators,
- 35 (2) the character sequence does not cross a record boundary,
- 36 (3) the first nonblank character is not a quotation mark or an apostrophe,
- 37 (4) the leading characters are not *digits* followed by an asterisk, and
- 38 (5) the character sequence contains at least one character,

39 the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the
 40 character sequence is terminated by the first blank, comma (if the decimal edit mode is POINT),
 41 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.

1 Let *len* be the length of the next effective item, and let *w* be the length of the character sequence. If
 2 *len* is less than or equal to *w*, the leftmost *len* characters of the sequence are transmitted to the next
 3 effective item. If *len* is greater than *w*, the sequence is transmitted to the leftmost *w* characters of the
 4 next effective item and the remaining *len*−*w* characters of the next effective item are filled with blanks.
 5 The effect is as though the sequence were assigned to the next effective item in an intrinsic assignment
 6 statement (7.2.1.3).

7 10.10.3.1 Null values

8 A null value is specified by

- 9 (1) the *r** form,
- 10 (2) no characters between consecutive value separators, or
- 11 (3) no characters before the first value separator in the first record read by each execution of a
 12 list-directed input statement.

NOTE 10.30

The end of a record following any other value separator, with or without separating blanks, does not specify a null value in list-directed input.

13 A null value has no effect on the definition status of the next effective item. A null value shall not be
 14 used for either the real or imaginary part of a complex constant, but a single null value may represent
 15 an entire complex constant.

16 A slash encountered as a value separator during execution of a list-directed input statement causes
 17 termination of execution of that input statement after the transference of the previous value. Any
 18 characters remaining in the current record are ignored. If there are additional items in the input list, the
 19 effect is as if null values had been supplied for them. Any *do-variable* in the input list becomes defined
 20 as if enough null values had been supplied for any remaining input list items.

NOTE 10.31

All blanks in a list-directed input record are considered to be part of some value separator except for

- (1) blanks embedded in a character sequence,
- (2) embedded blanks surrounding the real or imaginary part of a complex constant, and
- (3) leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma.

NOTE 10.32

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$
```

NOTE 10.32 (cont.)

The results are:

Variable	Value
I	12345
X (1)	12345.0
X (2)	unchanged
X (3)	1.5
X (4)	1.5
X (5) – X (8)	unchanged
P	ISN'T_BOB'S
Z	(123.0,0.0)
G	true

10.10.4 List-directed output

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.

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.

Bits output constants are produced with the effect of a *Zw.m* edit descriptor with *w* and *m* equal to $\text{CEILING}(k/4.0)$ where *k* is the kind type parameter value of the list item.

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 *0PFw.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 may occur between the separator and the imaginary part only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within a complex constant are between the separator and the end of a record and one blank at the beginning of the next record.

Character sequences produced when the delimiter mode has a value of NONE

- (1) are not delimited by apostrophes or quotation marks,
- (2) are not separated from each other by value separators,
- (3) have each internal apostrophe or quotation mark represented externally by one apostrophe or quotation mark, and
- (4) 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.

- 1 Character sequences produced when the delimiter mode has a value of QUOTE are delimited by quotes,
 2 are preceded and followed by a value separator, and have each internal quote represented on the external
 3 medium by two contiguous quotes.
- 4 Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited
 5 by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe
 6 represented on the external medium by two contiguous apostrophes.
- 7 If two or more successive values in an output record have identical values, the processor has the option
 8 of producing a repeated constant of the form $r*c$ instead of the sequence of identical values.
- 9 Slashes, as value separators, and null values are not produced as output by list-directed formatting.
- 10 Except for continuation of delimited character sequences, each output record begins with a blank char-
 11 acter.

NOTE 10.33

The length of the output records is not specified and may be processor dependent.

12 10.11 Namelist formatting

13 10.11.1 General

14 Namelist input/output allows data editing with NAME=value subsequences. This facilitates documen-
 15 tation of input and output files and more flexibility on input.

16 10.11.2 Name-value subsequences

17 The characters in one or more namelist records constitute a sequence of **name-value subsequences**,
 18 each of which consists of an object designator followed by an equals and followed by one or more values
 19 and value separators. The equals may optionally be preceded or followed by one or more contiguous
 20 blanks. The end of a record has the same effect as a blank character, unless it is within a character
 21 constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within
 22 a character constant.

23 The name may be any name in the *namelist-group-object-list* (5.6).

24 A value separator for namelist formatting is the same as for list-directed formatting (10.10).

25 10.11.3 Namelist input

26 Input for a namelist input statement consists of

- 27 (1) optional blanks and namelist comments,
- 28 (2) the character & followed immediately by the *namelist-group-name* as specified in the NAME-
 29 LIST statement,
- 30 (3) one or more blanks,
- 31 (4) a sequence of zero or more name-value subsequences separated by value separators, and
- 32 (5) a slash to terminate the namelist input.

NOTE 10.34

A slash encountered in a namelist input record causes the input statement to terminate. A slash cannot be used to separate two values in a namelist input statement.

1 In each name-value subsequence, the name shall be the name of a namelist group object list item with
2 an optional qualification and the name with the optional qualification shall not be a zero-sized array, a
3 zero-sized array section, or a zero-length character string. The optional qualification, if any, shall not
4 contain a vector subscript.

5 A group name or object name is without regard to case.

6 **10.11.3.1 Namelist group object names**

7 Within the input data, each name shall correspond to a particular namelist group object name. Sub-
8 scripts, strides, and substring range expressions used to qualify group object names shall be optionally
9 signed integer literal constants with no kind type parameters specified. If a namelist group object is
10 an array, the input record corresponding to it may contain either the array name or the designator of
11 a subobject of that array, using the syntax of object designators (R603). If the namelist group object
12 name is the name of a variable of derived type, the name in the input record may be either the name of
13 the variable or the designator of one of its components, indicated by qualifying the variable name with
14 the appropriate component name. Successive qualifications may be applied as appropriate to the shape
15 and type of the variable represented.

16 The order of names in the input records need not match the order of the namelist group object items.
17 The input records need not contain all the names of the namelist group object items. The definition
18 status of any names from the *namelist-group-object-list* that do not occur in the input record remains
19 unchanged. In the input record, each object name or subobject designator may be preceded and followed
20 by one or more optional blanks but shall not contain embedded blanks.

21 **10.11.3.2 Namelist group object list items**

22 The name-value subsequences are evaluated serially, in left-to-right order. A namelist group object
23 designator may appear in more than one name-value sequence.

24 When the name in the input record represents an array variable or a variable of derived type, the effect
25 is as if the variable represented were expanded into a sequence of scalar list items, in the same way that
26 formatted input/output list items are expanded (9.5.3). Each input value following the equals shall then
27 be acceptable to format specifications for the type of the list item in the corresponding position in the
28 expanded sequence, except as noted in this subclause. The number of values following the equals shall
29 not exceed the number of list items in the expanded sequence, but may be less; in the latter case, the
30 effect is as if sufficient null values had been appended to match any remaining list items in the expanded
31 sequence.

NOTE 10.35

For example, if the name in the input record is the name of an integer array of size 100, at most 100 values, each of which is either a digit string or a null value, may follow the equals; these values would then be assigned to the elements of the array in array element order.

32 A slash encountered as a value separator during the execution of a namelist input statement causes
33 termination of execution of that input statement after transference of the previous value. If there are
34 additional items in the namelist group object being transferred, the effect is as if null values had been
35 supplied for them.

36 A namelist comment may appear after any value separator except a slash. A namelist comment is also
37 permitted to start in the first nonblank position of an input record except within a character literal
38 constant.

39 Successive namelist records are read by namelist input until a slash is encountered; the remainder of the
40 record is ignored and need not follow the rules for namelist input values.

1 10.11.3.3 Namelist input values

2 Each value is either a null value (10.11.3.4), c , r^*c , or r^* , where c is a literal constant, optionally signed
 3 if integer or real, and r is an unsigned, nonzero, integer literal constant. A kind type parameter shall not
 4 be specified for c or r . The constant c is interpreted as though it had the same kind type parameter as
 5 the corresponding effective item. The r^*c form is equivalent to r successive appearances of the constant
 6 c , and the r^* form is equivalent to r successive null values. Neither of these forms may contain embedded
 7 blanks, except where permitted within the constant c .

8 The datum c (10.11) is any input value acceptable to format specifications for a given type, except for
 9 a restriction on the form of input values corresponding to list items of types logical, integer, bits, and
 10 character as specified in this subclause. The form of a real or complex value is dependent on the decimal
 11 edit mode in effect (10.6). The form of an input value shall be acceptable for the type of the namelist
 12 group object list item. The number and forms of the input values that may follow the equals in a name-
 13 value subsequence depend on the shape and type of the object represented by the name in the input
 14 record. When the name in the input record is that of a scalar variable of an intrinsic type, the equals
 15 shall not be followed by more than one value. Blanks are never used as zeros, and embedded blanks are
 16 not permitted in constants except within character constants and complex constants as specified in this
 17 subclause.

18 When the next effective namelist group object list item is of type real, the input form of the input value
 19 is that of a numeric input field. A numeric input field is a field suitable for F editing (10.7.2.3.2) that is
 20 assumed to have no fractional digits unless a decimal symbol appears within the field.

21 When the next effective item is of type complex, the input form of the input value consists of a left
 22 parenthesis followed by an ordered pair of numeric input fields separated by a comma (if the decimal
 23 edit mode is POINT) or a semicolon (if the decimal edit mode is COMMA), and followed by a right
 24 parenthesis. The first numeric input field is the real part of the complex constant and the second part
 25 is the imaginary part. Each of the numeric input fields may be preceded or followed by any number of
 26 blanks and ends of records. The end of a record may occur between the real part and the comma or
 27 semicolon, or between the comma or semicolon and the imaginary part.

28 When the next effective item is of type logical, the input form of the input value shall not include equals
 29 or value separators among the optional characters permitted for L editing (10.7.3).

30 When the next effective item is of type integer, the value in the input record is interpreted as if an Iw
 31 edit descriptor with a suitable value of w were used.

32 When the next effective item is of type bits, the value in the input record is interpreted as if a Zw edit
 33 descriptor with a suitable value of w were used.

34 When the next effective item is of type character, the input form consists of a delimited sequence of zero
 35 or more *rep-chars* whose kind type parameter is implied by the kind of the corresponding list item. Such
 36 a sequence may be continued from the end of one record to the beginning of the next record, but the
 37 end of record shall not occur between a doubled apostrophe in an apostrophe-delimited sequence, nor
 38 between a doubled quote in a quote-delimited sequence. The end of the record does not cause a blank or
 39 any other character to become part of the sequence. The sequence may be continued on as many records
 40 as needed. The characters blank, comma, semicolon, and slash may appear in such character sequences.

NOTE 10.36

A character sequence corresponding to a namelist input item of character type shall be delimited either with apostrophes or with quotes. The delimiter is required to avoid ambiguity between undelimited character sequences and object names. The value of the DELIM= specifier, if any, in the OPEN statement for an external file is ignored during namelist input (9.4.5.6).

1 Let *len* be the length of the next effective item, and let *w* be the length of the character sequence. If
 2 *len* is less than or equal to *w*, the leftmost *len* characters of the sequence are transmitted to the next
 3 effective item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the
 4 next effective item and the remaining *len*−*w* characters of the next effective item are filled with blanks.
 5 The effect is as though the sequence were assigned to the next effective item in an intrinsic assignment
 6 statement (7.2.1.3).

7 10.11.3.4 Null values

8 A null value is specified by

- 9 (1) the *r** form,
- 10 (2) blanks between two consecutive nonblank value separators following an equals,
- 11 (3) zero or more blanks preceding the first value separator and following an equals, or
- 12 (4) two consecutive nonblank value separators.

13 A null value has no effect on the definition status of the corresponding input list item. If the namelist
 14 group object list item is defined, it retains its previous value; if it is undefined, it remains undefined. A
 15 null value shall not be used as either the real or imaginary part of a complex constant, but a single null
 16 value may represent an entire complex constant.

NOTE 10.37

The end of a record following a value separator, with or without intervening blanks, does not specify a null value in namelist input.

17 10.11.3.5 Blanks

18 All blanks in a namelist input record are considered to be part of some value separator except for

- 19 (1) blanks embedded in a character constant,
- 20 (2) embedded blanks surrounding the real or imaginary part of a complex constant,
- 21 (3) leading blanks following the equals unless followed immediately by a slash or comma, or a
 22 semicolon if the decimal edit mode is comma, and
- 23 (4) blanks between a name and the following equals.

24 10.11.3.6 Namelist Comments

25 Except within a character literal constant, a “!” character after a value separator or in the first nonblank
 26 position of a namelist input record initiates a comment. The comment extends to the end of the current
 27 input record and may contain any graphic character in the processor-dependent character set. The
 28 comment is ignored. A slash within the namelist comment does not terminate execution of the namelist
 29 input statement. Namelist comments are not allowed in stream input because comments depend on
 30 record structure.

NOTE 10.38

Namelist input example:

```
INTEGER I; REAL X (8); CHARACTER (11) P; COMPLEX Z;
LOGICAL G
NAMELIST / TODAY / G, I, P, Z, X
READ (*, NML = TODAY)
```

The input data records are:

NOTE 10.38 (cont.)

```
&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

1 10.11.4 Namelist output

2 The form of the output produced is the same as that required for input, except for the forms of real,
 3 character, and logical values. The name in the output is in upper case. With the exception of adjacent
 4 undelimited character values, the values are separated by one or more blanks or by a comma, or a
 5 semicolon if the decimal edit mode is COMMA, optionally preceded by one or more blanks and optionally
 6 followed by one or more blanks.

7 Namelist output shall not include namelist comments.

8 The processor may begin new records as necessary. However, except for complex constants and character
 9 values, the end of a record shall not occur within a constant, character value, or name, and blanks shall
 10 not appear within a constant, character value, or name.

NOTE 10.39

The length of the output records is not specified exactly and may be processor dependent.

11 10.11.4.1 Namelist output editing

12 Values in namelist output records are edited as for list-directed output (10.10.4).

NOTE 10.40

Namelist output records produced with a DELIM= specifier with a value of NONE and which contain a character sequence might not be acceptable as namelist input records.

13 10.11.4.2 Namelist output records

14 If two or more successive values for the same namelist group item in an output record produced have
 15 identical values, the processor has the option of producing a repeated constant of the form r^*c instead
 16 of the sequence of identical values.

17 The name of each namelist group object list item is placed in the output record followed by an equals
 18 and a list of values of the namelist group object list item.

19 An ampersand character followed immediately by a *namelist-group-name* will be produced by namelist
 20 formatting at the start of the first output record to indicate which particular group of data objects is
 being output. A slash is produced by namelist formatting to indicate the end of the namelist formatting.

- 1 A null value is not produced by namelist formatting.
- 2 Except for new records created by explicit formatting within a user-defined derived-type output pro-
- 3 cedure or by continuation of delimited character sequences, each output record begins with a blank
- 4 character.

11 Program units

11.1 Main program

A Fortran **main program** is a program unit that does not contain a SUBROUTINE, FUNCTION, MODULE, SUBMODULE, or BLOCK DATA statement as its first statement.

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

```
R1102  program-stmt           is  PROGRAM program-name
```

```
R1103  end-program-stmt       is  END [ PROGRAM [ program-name ] ]
```

C1101 (R1101) In a *main-program*, the *execution-part* shall not contain a RETURN statement or an ENTRY statement.

C1102 (R1101) The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, shall be identical to the *program-name* specified in the *program-stmt*.

NOTE 11.1

The program name is global to the program (16.2). For explanatory information about uses for the program name, see subclause C.8.1.

NOTE 11.2

An example of a main program is:

```
PROGRAM ANALYZE
  REAL A, B, C (10,10)      ! Specification part
  CALL FIND                 ! Execution part
CONTAINS
  SUBROUTINE FIND           ! Internal subprogram
    ...
  END SUBROUTINE FIND
END PROGRAM ANALYZE
```

The main program may be defined by means other than Fortran; in that case, the program shall not contain a *main-program* program unit.

A reference to a Fortran *main-program* shall not appear in any program unit in the program, including itself.

11.2 Modules

1 11.2.1 General

2 A **module** contains specifications and definitions that are to be accessible to other program units by use
 3 association. A module that is provided as an inherent part of the processor is an **intrinsic module**. A
 4 **nonintrinsic module** is defined by a module program unit or a means other than Fortran.

5 Procedures and types defined in an intrinsic module are not themselves intrinsic.

6 R1104 *module* **is** *module-stmt*
 7 *[specification-part]*
 8 *[module-subprogram-part]*
 9 *end-module-stmt*

10 R1105 *module-stmt* **is** **MODULE** *module-name*

11 R1106 *end-module-stmt* **is** **END** [**MODULE** [*module-name*]]

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

14 R1108 *module-subprogram* **is** *function-subprogram*
 15 **or** *subroutine-subprogram*
 16 **or** *separate-module-subprogram*

17 C1103 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the
 18 *module-name* specified in the *module-stmt*.

19 C1104 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a
 20 *format-stmt*.

21 C1105 (R1104) If an object that has a default-initialized direct component is declared in the specifica-
 22 tion part of a module it shall have the **ALLOCATABLE**, **POINTER**, or **SAVE** attribute.

23 C1106 If an object that has a co-array ultimate component is declared in the *specification-part* of a
 24 module it shall have the **SAVE** attribute.

J3 internal note

Unresolved Technical Issue 010

This is better, but still inconsistent.

After 06-238r1, top-level allocatable co-arrays are allowed in a module without being **SAVED**. Allow that but requiring **SAVE** for things with co-array components does not make sense.

06-238r1 claims “save is required for a module variable if it has a co-array component. We wish to retain this since otherwise an unexpected implicit synchronization would be needed whenever the module ceases to be active.” Why would that be unexpected? I would expect that when the module goes out of scope, the allocatable (co-array thingo) either remains (as is allowed) or is deallocated (as is allowed), with only the latter case requiring (implicit) synchronisation.

NOTE 11.3

The module name is global to the program (16.2).

NOTE 11.4

Although statement function definitions, **ENTRY** statements, and **FORMAT** statements shall not appear in the specification part of a module, they may appear in the specification part of a module subprogram in the module.

NOTE 11.5

For a discussion of the impact of modules on dependent compilation, see subclause C.8.2.

NOTE 11.6

For examples of the use of modules, see subclause C.8.3.

1 If a procedure declared in the scoping unit of a module has an implicit interface, it shall be given the
 2 EXTERNAL attribute in that scoping unit; if it is a function, its type and type parameters shall be
 3 explicitly declared in a type declaration statement in that scoping unit.

4 If an intrinsic procedure is declared in the scoping unit of a module, it shall explicitly be given the
 5 INTRINSIC attribute in that scoping unit or be used as an intrinsic procedure in that scoping unit.

6 11.2.2 The USE statement and use association

7 The **USE statement** specifies use association. A USE statement is a **reference** to the module it
 8 specifies. At the time a USE statement is processed, the public portions of the specified module shall
 9 be available. A module shall not reference itself, either directly or indirectly. A submodule shall not
 10 reference its ancestor module by use association, either directly or indirectly.

NOTE 11.7

It is possible for submodules with different ancestor modules to reference each others' ancestor modules by use association.

11 The **USE statement** provides the means by which a scoping unit accesses named data objects, derived
 12 types, interface blocks, procedures, abstract interfaces, module procedure interfaces, generic identifiers,
 13 macros, and namelist groups in a module. The entities in the scoping unit are **use associated** with the
 14 entities in the module. The accessed entities have the attributes specified in the module, except that an
 15 entity may have a different accessibility attribute or it may have the ASYNCHRONOUS or VOLATILE
 16 attribute in the local scoping unit even if the associated module entity does not. The entities made
 17 accessible are identified by the names or generic identifiers used to identify them in the module. By
 18 default, they are identified by the same identifiers in the scoping unit containing the USE statement,
 19 but it is possible to specify that different local identifiers are used.

NOTE 11.8

The accessibility of module entities may be controlled by accessibility attributes (4.5.2.2, 5.3.2), and the ONLY option of the USE statement. Definability of module entities can be controlled by the PROTECTED attribute (5.3.14).

20	R1109	<i>use-stmt</i>	is	USE [[, <i>module-nature</i>] ::] <i>module-name</i> [, <i>rename-list</i>]
21			or	USE [[, <i>module-nature</i>] ::] <i>module-name</i> , ■
22				■ ONLY : [<i>only-list</i>]
23	R1110	<i>module-nature</i>	is	INTRINSIC
24			or	NON-INTRINSIC
25	R1111	<i>rename</i>	is	<i>local-name</i> => <i>use-name</i>
26			or	OPERATOR (<i>local-defined-operator</i>) => ■
27				■ OPERATOR (<i>use-defined-operator</i>)
28	R1112	<i>only</i>	is	<i>generic-spec</i>
29			or	<i>only-use-name</i>

- 1 **or** *rename*
- 2 R1113 *only-use-name* **is** *use-name*
- 3 C1107 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.
- 4 C1108 (R1109) If *module-nature* is NON_INTRINSIC, *module-name* shall be the name of a nonintrinsic
- 5 module.
- 6 C1109 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the
- 7 same name.
- 8 C1110 (R1111) OPERATOR(*use-defined-operator*) shall not identify a type-bound generic interface.
- 9 C1111 (R1112) The *generic-spec* shall not identify a type-bound generic interface.

NOTE 11.9

The above two constraints 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*.

- 10 C1112 (R1112) Each *generic-spec* shall be a public entity in the module.
- 11 C1113 (R1113) Each *use-name* shall be the name of a public entity in the module.
- 12 R1114 *local-defined-operator* **is** *defined-unary-op*
- 13 **or** *defined-binary-op*
- 14 R1115 *use-defined-operator* **is** *defined-unary-op*
- 15 **or** *defined-binary-op*
- 16 C1114 (R1115) Each *use-defined-operator* shall be a public entity in the module.
- 17 A *use-stmt* without a *module-nature* provides access either to an intrinsic or to a nonintrinsic module.
- 18 If the *module-name* is the name of both an intrinsic and a nonintrinsic module, the nonintrinsic module
- 19 is accessed.
- 20 The USE statement without the ONLY option provides access to all public entities in the specified
- 21 module.
- 22 A USE statement with the ONLY option provides access only to those entities that appear as *generic-*
- 23 *specs*, *use-names*, or *use-defined-operators* in the *only-list*.
- 24 More than one USE statement for a given module may appear in a scoping unit. If one of the USE
- 25 statements is without an ONLY option, all public entities in the module are accessible. If all the USE
- 26 statements have ONLY options, only those entities in one or more of the *only-lists* are accessible.
- 27 An accessible entity in the referenced module has one or more local identifiers. These identifiers are
- 28 (1) the identifier of the entity in the referenced module if that identifier appears as an *only-use-*
- 29 *name* or as the *defined-operator* of a *generic-spec* in any *only* for that module,
- 30 (2) each of the *local-names* or *local-defined-operators* that the entity is given in any *rename* for
- 31 that module, and
- 32 (3) the identifier of the entity in the referenced module if that identifier does not appear as a
- 33 *use-name* or *use-defined-operator* in any *rename* for that module.
- 34 Two or more accessible entities, other than generic interfaces or defined operators, may have the same
- 35 identifier only if the identifier is not used to refer to an entity in the scoping unit. Generic interfaces
- 36 and defined operators are handled as described in 12.4.3.4 and 12.4.3.4.5. Except for these cases, the

- 1 local identifier of any entity given accessibility by a USE statement shall differ from the local identifiers
 2 of all other entities accessible to the scoping unit through USE statements and otherwise.

NOTE 11.10

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.

- 3 The local identifier of an entity made accessible by a USE statement shall not appear in any other
 4 nonexecutable statement that would cause any attribute (5.3) of the entity to be specified in the scoping
 5 unit that contains the USE statement, except that it may appear in a PUBLIC or PRIVATE statement
 6 in the scoping unit of a module and it may be given the ASYNCHRONOUS or VOLATILE attribute.
- 7 The appearance of such a local identifier in a PUBLIC statement in a module causes the entity accessible
 8 by the USE statement to be a public entity of that module. If the identifier appears in a PRIVATE
 9 statement in a module, the entity is not a public entity of that module. If the local identifier does not
 10 appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (5.4.1)
 11 of that scoping unit.

NOTE 11.11

The constraints in subclauses 5.7.1, 5.7.2, and 5.6 prohibit the *local-name* from appearing as a *common-block-object* in a COMMON statement, an *equivalence-object* in an EQUIVALENCE statement, or a *namelist-group-name* in a NAMELIST statement, respectively. There is no prohibition against the *local-name* appearing as a *common-block-name* or a *namelist-group-object*.

NOTE 11.12

For a discussion of the impact of the ONLY option and renaming on dependent compilation, see subclause C.8.2.1.

NOTE 11.13

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS_LIB.

```
USE MATH_LIB; USE STATS_LIB, SPROD => PROD
```

makes all public entities in both MATH_LIB and STATS_LIB accessible. If MATH_LIB contains an entity called PROD, it is accessible by its own name while the entity PROD of STATS_LIB is accessible by the name SPROD.

```
USE STATS_LIB, ONLY: YPROD; USE STATS_LIB, ONLY : PROD
```

makes public entities YPROD and PROD in STATS_LIB accessible.

```
USE STATS_LIB, ONLY : YPROD; USE STATS_LIB
```

makes all public entities in STATS_LIB accessible.

12 **11.2.3 Submodules**

1 A **submodule** is a program unit that extends a module or another submodule. The program unit that
 2 it extends is its **parent**, and is specified by the *parent-identifier* in the *submodule-stmt*. A submodule
 3 is a **child** of its parent. An **ancestor** of a submodule is its parent or an ancestor of its parent. A
 4 **descendant** of a module or submodule is one of its children or a descendant of one of its children. The
 5 **submodule identifier** is the ordered pair whose first element is the ancestor module name and whose
 6 second element is the submodule name.

NOTE 11.14

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 may optionally have one or more ancestor submodules.

7 A submodule accesses the scoping unit of its parent by host association (16.5.1.4).

8 A submodule may provide implementations for module procedures, each of which is declared by a module
 9 procedure interface body (12.4.3.2) within that submodule or one of its ancestors, and declarations and
 10 definitions of other entities that are accessible by host association in its descendants.

11 R1116 *submodule* **is** *submodule-stmt*
 12 [*specification-part*]
 13 [*module-subprogram-part*]
 14 *end-submodule-stmt*

15 R1117 *submodule-stmt* **is** SUBMODULE (*parent-identifier*) *submodule-name*

16 R1118 *parent-identifier* **is** *ancestor-module-name* [: *parent-submodule-name*]

17 R1119 *end-submodule-stmt* **is** END [SUBMODULE [*submodule-name*]]

18 C1115 (R1116) A submodule *specification-part* shall not contain a *format-stmt*, *entry-stmt*, or *stmt-*
 19 *function-stmt*.

20 C1116 (R1116) An object with a default-initialized direct component that is declared in the specification
 21 part of a submodule shall have the ALLOCATABLE, POINTER, or SAVE attribute.

22 C1117 (R1118) The *ancestor-module-name* shall be the name of a nonintrinsic module; the *parent-*
 23 *submodule-name* shall be the name of a descendant of that module.

24 C1118 (R1116) If a *submodule-name* appears in the *end-submodule-stmt*, it shall be identical to the one
 25 in the *submodule-stmt*.

26 11.3 Block data program units

27 A **block data program unit** is used to provide initial values for data objects in named common blocks.

28 R1120 *block-data* **is** *block-data-stmt*
 29 [*specification-part*]
 30 *end-block-data-stmt*

31 R1121 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

32 R1122 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

33 C1119 (R1120) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided
 34 in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-*
 35 *data-stmt*.

1 C1120 (R1120) A *block-data specification-part* shall contain only derived-type definitions and ASYN-
 2 CHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRIN-
 3 SIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration
 4 statements.

5 C1121 (R1120) A type declaration statement in a *block-data specification-part* shall not contain AL-
 6 LOCATABLE, EXTERNAL, or BIND attribute specifiers.

NOTE 11.15

For explanatory information about the uses for the *block-data-name*, see subclause C.8.1.

7 If an object in a named common block is initially defined, all storage units in the common block storage
 8 sequence shall be specified even if they are not all initially defined. More than one named common block
 9 may have objects initially defined in a single block data program unit.

NOTE 11.16

In the example

```
BLOCK DATA INIT
  REAL A, B, C, D, E, F
  COMMON /BLOCK1/ A, B, C, D
  DATA A /1.2/, C /2.3/
  COMMON /BLOCK2/ E, F
  DATA F /6.5/
END BLOCK DATA INIT
```

common blocks BLOCK1 and BLOCK2 both have objects that are being initialized in a single block data program unit. B, D, and E are not initialized but they need to be specified as part of the common blocks.

10 Only an object in a named common block may be initially defined in a block data program unit.

NOTE 11.17

Objects associated with an object in a common block are considered to be in that common block.

11 The same named common block shall not be specified in more than one block data program unit in a
 12 program.

13 There shall not be more than one unnamed block data program unit in a program.

NOTE 11.18

An example of a block data program unit is:

```
BLOCK DATA WORK
  COMMON /WRKCOM/ A, B, C (10, 10)
  REAL :: A, B, C
  DATA A /1.0/, B /2.0/, C /100 * 0.0/
END BLOCK DATA WORK
```


1 12 Procedures

2 12.1 Concepts

3 The concept of a procedure was introduced in 2.2.4. This clause contains a complete description of
4 procedures. The actions specified by a procedure are performed when the procedure is invoked by
5 execution of a reference to it.

6 The sequence of actions encapsulated by a procedure has access to entities in the invoking scoping
7 unit by way of argument association (12.5.2). A **dummy argument** is a name that appears in the
8 SUBROUTINE, FUNCTION, or ENTRY statement in the declaration of a procedure (R1235). Dummy
9 arguments are also specified for intrinsic procedures and procedures in intrinsic modules in Clauses 13,
10 14, and 15.

11 The entities in the invoking scoping unit are specified by actual arguments. An **actual argument** is an
12 entity that appears in a procedure reference (R1223).

13 A procedure may also have access to entities in other scoping units, not necessarily the invoking scoping
14 unit, by use association (16.5.1.3), host association (16.5.1.4), linkage association (16.5.1.5), storage
15 association (16.5.3), or by reference to external procedures (5.3.8).

16 12.2 Procedure classifications

17 12.2.1 Procedure classification by reference

18 The definition of a procedure specifies it to be a function or a subroutine. A reference to a function
19 either appears explicitly as a primary within an expression, or is implied by a defined operation (7.1.6)
20 within an expression. A reference to a subroutine is a CALL statement, a defined assignment statement
21 (7.2.1.4), the appearance of an object processed by user-defined derived-type input/output (9.5.4.7) in
22 an input/output list, or finalization (4.5.6).

23 A procedure is classified as **elemental** if it is a procedure that may be referenced elementally (12.8).

24 12.2.2 Procedure classification by means of definition

25 12.2.2.1 Intrinsic procedures

26 A procedure that is provided as an inherent part of the processor is an **intrinsic procedure**.

27 12.2.2.2 External, internal, and module procedures

28 An **external procedure** is a procedure that is defined by an external subprogram or by a means other
29 than Fortran.

30 An **internal procedure** is a procedure that is defined by an internal subprogram. Internal subprograms
31 may appear in the main program, in an external subprogram, or in a module subprogram. Internal
32 subprograms shall not appear in other internal subprograms. Internal subprograms are the same as
33 external subprograms except that the name of the internal procedure is not a global identifier, an
34 internal subprogram shall not contain an ENTRY statement, and the internal subprogram has access to
35 host entities by host association.

1 A **module procedure** is a procedure that is defined by a module subprogram.

2 A subprogram defines a procedure for the SUBROUTINE or FUNCTION statement. If the subprogram
3 has one or more ENTRY statements, it also defines a procedure for each of them.

4 **12.2.2.3 Dummy procedures**

5 A dummy argument that is specified to be a procedure or appears in a procedure reference is a **dummy**
6 **procedure**. A dummy procedure with the POINTER attribute is a dummy procedure pointer.

7 **12.2.2.4 Procedure pointers**

8 A procedure pointer is a procedure that has the EXTERNAL and POINTER attributes; it may be
9 pointer associated with an external procedure, an internal procedure, an intrinsic procedure, a module
10 procedure, or a dummy procedure that is not a procedure pointer.

11 **12.2.2.5 Statement functions**

12 A function that is defined by a single statement is a **statement function** (12.6.4).

13 **12.3 Characteristics**

14 **12.3.1 Characteristics of procedures**

15 The **characteristics of a procedure** are the classification of the procedure as a function or subroutine,
16 whether it is pure, whether it is elemental, whether it has the BIND attribute, the characteristics of its
17 dummy arguments, and the characteristics of its result value if it is a function.

18 **12.3.2 Characteristics of dummy arguments**

19 **12.3.2.1 General**

20 Each dummy argument has the characteristic that it is a dummy data object, a dummy procedure, a
21 dummy procedure pointer, or an asterisk (alternate return indicator). A dummy argument other than an asterisk
22 may be specified to have the OPTIONAL attribute. This attribute means that the dummy argument
23 need not be associated with an actual argument for any particular reference to the procedure.

24 **12.3.2.2 Characteristics of dummy data objects**

25 The characteristics of a dummy data object are its type, its type parameters (if any), its shape, its
26 co-rank, its co-dimensions, its intent (5.3.9, 5.4.8), whether it is optional (5.3.11, 5.4.9), whether it is
27 allocatable (5.3.7.4), whether it has the ASYNCHRONOUS (5.3.4), CONTIGUOUS (5.3.6), VALUE
28 (5.3.17), or VOLATILE (5.3.18) attributes, whether it is polymorphic, and whether it is a pointer
29 (5.3.13, 5.4.11) or a target (5.3.16, 5.4.14). If a type parameter of an object or a bound of an array is not
30 an initialization expression, the exact dependence on the entities in the expression is a characteristic. If
31 a shape, size, or type parameter is assumed or deferred, it is a characteristic.

32 **12.3.2.3 Characteristics of dummy procedures and dummy procedure pointers**

33 The characteristics of a dummy procedure are the explicitness of its interface (12.4.2), its characteristics
34 as a procedure if the interface is explicit, whether it is a pointer, and whether it is optional (5.3.11, 5.4.9).

35 **12.3.2.4 Characteristics of asterisk dummy arguments**

36 An asterisk as a dummy argument has no characteristics.

1 12.3.3 Characteristics of function results

2 The characteristics of a function result are its type, type parameters (if any), rank, whether it is poly-
 3 morphic, whether it is allocatable, whether it is a pointer, whether it has the CONTIGUOUS attribute,
 4 and whether it is a procedure pointer. If a function result is an array that is not allocatable or a pointer,
 5 its shape is a characteristic. If a type parameter of a function result or a bound of a function result
 6 array is not an initialization expression, the exact dependence on the entities in the expression is a
 7 characteristic. If type parameters of a function result are deferred, which parameters are deferred is a
 8 characteristic. If the length of a character function result is assumed, this is a characteristic.

9 12.4 Procedure interface

10 12.4.1 General

11 The **interface** of a procedure determines the forms of reference through which it may be invoked.
 12 The procedure's interface consists of its abstract interface, its name, its binding label if any, and the
 13 procedure's generic identifiers, if any. The characteristics of a procedure are fixed, but the remainder
 14 of the interface may differ in different scoping units, except that for a separate module procedure body
 15 (12.6.2.5), the dummy argument names, binding label, and whether it is recursive shall be the same as
 16 in its corresponding module procedure interface body (12.4.3.2).

17 An **abstract interface** consists of procedure characteristics and the names of dummy arguments.

NOTE 12.1

For more explanatory information on procedure interfaces, see C.9.3.

18 12.4.2 Implicit and explicit interfaces

19 If a procedure is accessible in a scoping unit, its interface is either **explicit** or **implicit** in that scoping
 20 unit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit
 21 in such a scoping unit. The interface of a subroutine or a function with a separate result name is explicit
 22 within the subprogram that defines it. The interface of a statement function is always implicit. The interface of
 23 an external procedure or dummy procedure is explicit in a scoping unit other than its own if an interface
 24 body (12.4.3.2) for the procedure is supplied or accessible, and implicit otherwise.

NOTE 12.2

For example, the subroutine LLS of C.8.3.5 has an explicit interface.

25 12.4.2.1 Explicit interface

26 A procedure other than a statement function shall have an explicit interface if it is referenced and

- 27 (1) a reference to the procedure appears
 - 28 (a) with an argument keyword (12.5.2),
 - 29 (b) in a context that requires it to be pure, or
 - 30 (c) with an argument that it not of type bits corresponding to a dummy argument of
 31 type bits,
- 32 (2) the procedure has a dummy argument that
 - 33 (a) has the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, TARGET,
 34 VALUE, or VOLATILE attribute,
 - 35 (b) is an assumed-shape array,

- 1 (c) is a co-array,
- 2 (d) is of a parameterized derived type, or
- 3 (e) is polymorphic,
- 4 (3) the procedure has a result that
 - 5 (a) is an array,
 - 6 (b) is a pointer or is allocatable, or
 - 7 (c) has a nonassumed type parameter value that is not an initialization expression,
- 8 (4) the procedure is elemental, or
- 9 (5) the procedure has the BIND attribute.

10 12.4.3 Specification of the procedure interface

11 12.4.3.1 General

12 The interface for an internal, external, module, or dummy procedure is specified by a FUNCTION,
 13 SUBROUTINE, or ENTRY statement and by specification statements for the dummy arguments and
 14 the result of a function. These statements may appear in the procedure definition, in an interface body,
 15 or both, except that the ENTRY statement shall not appear in an interface body.

NOTE 12.3

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 may appear in a PROCEDURE statement in an interface block (12.4.3.2).

16 12.4.3.2 Interface block

17 R1201	<i>interface-block</i>	is	<i>interface-stmt</i> [<i>interface-specification</i>] ... <i>end-interface-stmt</i>
20 R1202	<i>interface-specification</i>	is	<i>interface-body</i>
21		or	<i>procedure-stmt</i>
22 R1203	<i>interface-stmt</i>	is	INTERFACE [<i>generic-spec</i>]
23		or	ABSTRACT INTERFACE
24 R1204	<i>end-interface-stmt</i>	is	END INTERFACE [<i>generic-spec</i>]
25 R1205	<i>interface-body</i>	is	<i>function-stmt</i> [<i>specification-part</i>] <i>end-function-stmt</i>
26		or	<i>subroutine-stmt</i> [<i>specification-part</i>] <i>end-subroutine-stmt</i>
27			
28			
29			
30			
31 R1206	<i>procedure-stmt</i>	is	[MODULE] PROCEDURE [::] <i>procedure-name-list</i>
32 R1207	<i>generic-spec</i>	is	<i>generic-name</i>
33		or	OPERATOR (<i>defined-operator</i>)
34		or	ASSIGNMENT (=)
35		or	<i>dtio-generic-spec</i>

- 1 R1208 *dtio-generic-spec* **is** READ (FORMATTED)
 2 **or** READ (UNFORMATTED)
 3 **or** WRITE (FORMATTED)
 4 **or** WRITE (UNFORMATTED)
- 5 C1201 (R1201) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure
 6 defined by that subprogram.
- 7 C1202 (R1201) The *generic-spec* shall be included in the *end-interface-stmt* only if it is provided in the
 8 *interface-stmt*. If the *end-interface-stmt* includes *generic-name*, the *interface-stmt* shall specify
 9 the same *generic-name*. If the *end-interface-stmt* includes ASSIGNMENT(=), the *interface-*
 10 *stmt* shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *dtio-generic-spec*, the
 11 *interface-stmt* shall specify the same *dtio-generic-spec*. If the *end-interface-stmt* includes OP-
 12 ERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If one
 13 *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the corre-
 14 sponding operator <, <=, >, >=, ==, or /=.
- 15 C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the
 16 *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword
 17 that specifies an intrinsic type.
- 18 C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.
- 19 C1205 (R1205) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments
 20 except pointer, alternate return, and procedure arguments.
- 21 C1206 (R1205) An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-*
 22 *function-stmt*.
- 23 C1207 (R1206) A *procedure-name* shall have an explicit interface and shall refer to an accessible pro-
 24 cedure pointer, external procedure, dummy procedure, or module procedure.
- 25 C1208 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall
 26 be accessible in the current scope as a module procedure.
- 27 C1209 (R1206) A *procedure-name* shall not specify a procedure that is specified previously in any
 28 *procedure-stmt* in any accessible interface with the same generic identifier.
- 29 An external or module subprogram specifies a **specific interface** for the procedures defined in that
 30 subprogram. Such a specific interface is explicit for module procedures and implicit for external proce-
 31 dures.
- 32 An interface block introduced by ABSTRACT INTERFACE is an **abstract interface block**. An
 33 interface body in an abstract interface block specifies an abstract interface. An interface block with a
 34 generic specification is a **generic interface block**. An interface block with neither ABSTRACT nor a
 35 generic specification is a **specific interface block**.
- 36 The name of the entity declared by an interface body is the *function-name* in the *function-stmt* or the
 37 *subroutine-name* in the *subroutine-stmt* that begins the interface body.
- 38 A **module procedure interface body** is an interface body whose initial statement contains the
 39 keyword MODULE. It defines the **module procedure interface** for a separate module procedure
 40 (12.6.2.5). A separate module procedure is accessible by use association if and only if its interface body
 41 is declared in the specification part of a module and is public. If a corresponding (12.6.2.5) separate
 42 module procedure is not defined, the interface may be used to specify an explicit specific interface but
 43 the procedure shall not be used in any other way.

- 1 C1210 (R1205) A module procedure interface body shall not appear in an abstract interface block.
- 2 An interface body in a generic or specific interface block specifies the `EXTERNAL` attribute and an
 3 explicit specific interface for an external procedure or a dummy procedure. If the name of the declared
 4 procedure is that of a dummy argument in the subprogram containing the interface body, the procedure
 5 is a dummy procedure; otherwise, it is an external procedure.
- 6 An interface body specifies all of the characteristics of the explicit specific interface or abstract interface.
 7 The specification part of an interface body may specify attributes or define values for data entities that
 8 do not determine characteristics of the procedure. Such specifications have no effect.
- 9 If an explicit specific interface is specified by an interface body or a procedure declaration statement
 10 (12.4.3.6) for an external procedure, the characteristics shall be consistent with those specified in the
 11 procedure definition, except that the interface may specify a procedure that is not pure if the procedure
 12 is defined to be pure. An interface for a procedure named by an `ENTRY` statement may be specified by
 13 using the entry name as the procedure name in the interface body. If an external procedure does not
 14 exist in the program, an interface body for it may be used to specify an explicit specific interface but
 15 the procedure shall not be used in any other way. An explicit specific interface may be specified by an
 16 interface body for an external procedure that does not exist in the program if the procedure is never
 17 used in any way. A procedure shall not have more than one explicit specific interface in a given scoping
 18 unit.

NOTE 12.4

The dummy argument names in an interface body may be different from the corresponding dummy argument names in the procedure definition because the name of a dummy argument is not a characteristic.

NOTE 12.5

An example of a specific interface block is:

```

INTERFACE
  SUBROUTINE EXT1 (X, Y, Z)
    REAL, DIMENSION (100, 100) :: X, Y, Z
  END SUBROUTINE EXT1
  SUBROUTINE EXT2 (X, Z)
    REAL X
    COMPLEX (KIND = 4) Z (2000)
  END SUBROUTINE EXT2
  FUNCTION EXT3 (P, Q)
    LOGICAL EXT3
    INTEGER P (1000)
    LOGICAL Q (1000)
  END FUNCTION EXT3
END INTERFACE

```

This interface block specifies explicit interfaces for the three external procedures `EXT1`, `EXT2`, and `EXT3`. Invocations of these procedures may use argument keywords (12.5.2); for example:

```
PRINT *, EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

19 **12.4.3.3 IMPORT statement**

- 20 R1209 *import-stmt* **is** `IMPORT [ [ :: ] import-name-list ]`

1 C1211 (R1209) The IMPORT statement is allowed only in an *interface-body* that is not a module
2 procedure interface body.

3 C1212 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.

4 The **IMPORT statement** specifies that the named entities from the host scoping unit are accessible in
5 the interface body by host association. An entity that is imported in this manner and is defined in the
6 host scoping unit shall be explicitly declared prior to the interface body. The name of an entity made
7 accessible by an IMPORT statement shall not appear in any of the contexts described in 16.5.1.4 that
8 cause the host entity of that name to be inaccessible.

9 Within an interface body, if an IMPORT statement with no *import-name-list* appears, each host entity
10 not named in an IMPORT statement also is made accessible by host association if its name does not
11 appear in any of the contexts described in 16.5.1.4 that cause the host entity of that name to be
12 inaccessible. If an entity that is made accessible by this means is accessed by host association and is
13 defined in the host scoping unit, it shall be explicitly declared prior to the interface body.

NOTE 12.6

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.

12.4.3.4 Generic interfaces

12.4.3.4.1 Generic identifiers

16 A generic interface block specifies a **generic interface** for each of the procedures in the interface block.
17 The **PROCEDURE statement** lists procedure pointers, external procedures, dummy procedures, or
18 module procedures that have this generic interface. A generic interface is always explicit.

19 The *generic-spec* in an *interface-stmt* is a **generic identifier** for all the procedures in the interface
20 block. The rules specifying how any two procedures with the same generic identifier shall differ are given

- 1 in 12.4.3.4.5. They ensure that any generic invocation applies to at most one specific procedure.
- 2 A **generic name** specifies a single name to reference all of the procedure names in the interface block.
- 3 A generic name may be the same as any one of the procedure names in the interface block, or the same
- 4 as any accessible generic name.
- 5 A generic name may be the same as a derived-type name, in which case all of the procedures in the
- 6 interface block shall be functions.
- 7 An *interface-stmt* having a *dtio-generic-spec* is an interface for a user-defined derived-type input/output
- 8 procedure (9.5.4.7).

NOTE 12.7

An example of a generic procedure interface is:

```
INTERFACE SWITCH
  SUBROUTINE INT_SWITCH (X, Y)
    INTEGER, INTENT (INOUT) :: X, Y
  END SUBROUTINE INT_SWITCH
  SUBROUTINE REAL_SWITCH (X, Y)
    REAL, INTENT (INOUT) :: X, Y
  END SUBROUTINE REAL_SWITCH
  SUBROUTINE COMPLEX_SWITCH (X, Y)
    COMPLEX, INTENT (INOUT) :: X, Y
  END SUBROUTINE COMPLEX_SWITCH
END INTERFACE SWITCH
```

Any of these three subroutines (INT_SWITCH, REAL_SWITCH, COMPLEX_SWITCH) may 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
```

9 **12.4.3.4.2 Defined operations**

- 10 If OPERATOR is specified in a generic specification, all of the procedures specified in the generic
- 11 interface shall be functions that may be referenced as defined operations (7.1.6, 12.5). In the case of
- 12 functions of two arguments, infix binary operator notation is implied. In the case of functions of one
- 13 argument, prefix operator notation is implied. OPERATOR shall not be specified for functions with no
- 14 arguments or for functions with more than two arguments. The dummy arguments shall be nonoptional
- 15 dummy data objects and shall be specified with INTENT (IN). The function result shall not have assumed
- 16 character length. If the operator is an *intrinsic-operator* (R310), the number of function arguments shall
- 17 be consistent with the intrinsic uses of that operator, and the types, kind type parameters, or ranks of
- 18 the dummy arguments shall differ from those required for the intrinsic operation (7.1.5).
- 19 A defined operation is treated as a reference to the function. For a unary defined operation, the operand
- 20 corresponds to the function's dummy argument; for a binary operation, the left-hand operand corre-
- 21 sponds to the first dummy argument of the function and the right-hand operand corresponds to the
- 22 second dummy argument.

NOTE 12.8

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR ( * )
```

NOTE 12.8 (cont.)

```

FUNCTION BOOLEAN_AND (B1, B2)
  LOGICAL, INTENT (IN) :: B1 (:), B2 (SIZE (B1))
  LOGICAL :: BOOLEAN_AND (SIZE (B1))
END FUNCTION BOOLEAN_AND
END INTERFACE OPERATOR ( * )

```

This allows, for example

```
SENSOR (1:N) * ACTION (1:N)
```

as an alternative to the function call

```

BOOLEAN_AND (SENSOR (1:N), ACTION (1:N))    ! SENSOR and ACTION are
                                              ! of type LOGICAL

```

- 1 A given defined operator may, as with generic names, apply to more than one function, in which case
- 2 it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic
- 3 properties include the intrinsic operations they represent. Because both forms of each relational operator
- 4 have the same interpretation (7.1.6.2), extending one form (such as <=) has the effect of defining both
- 5 forms (<= and .LE.).

NOTE 12.9

In Fortran 90 and Fortran 95, it was not possible to define operations on pointers because pointer dummy arguments were disallowed from having an INTENT attribute. The restriction against INTENT for pointer dummy arguments is now lifted, so defined operations on pointers are now possible.

However, the POINTER attribute cannot be used to resolve generic procedures (12.4.3.4.5), so it is not possible to define a generic operator that has one procedure for pointers and another procedure for nonpointers.

6 **12.4.3.4.3 Defined assignments**

- 7 If ASSIGNMENT (=) is specified in a generic specification, all the procedures in the generic interface
- 8 shall be subroutines that may be referenced as defined assignments (7.2.1.4). Defined assignment may,
- 9 as with generic names, apply to more than one subroutine, in which case it is generic in exact analogy
- 10 to generic procedure names.
- 11 Each of these subroutines shall have exactly two dummy arguments. The dummy arguments shall be
- 12 nonoptional dummy data objects. The first argument shall have INTENT (OUT) or INTENT (INOUT)
- 13 and the second argument shall have INTENT (IN). Either the second argument shall be an array whose
- 14 rank differs from that of the first argument, the declared types and kind type parameters of the arguments
- 15 shall not conform as specified in Table 7.12, or the first argument shall be of derived type. A defined
- 16 assignment is treated as a reference to the subroutine, with the left-hand side as the first argument
- 17 and the right-hand side enclosed in parentheses as the second argument. The ASSIGNMENT generic
- 18 specification specifies that assignment is extended or redefined.

NOTE 12.10

An example of the use of the ASSIGNMENT generic specification is:

```
INTERFACE ASSIGNMENT ( = )
```

NOTE 12.10 (cont.)

```

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'))

```

1 **12.4.3.4.4 User-defined derived-type input/output procedure interfaces**

2 All of the procedures specified in an interface block for a user-defined derived-type input/output proce-
 3 dure shall be subroutines that have interfaces as described in 9.5.4.7.2.

4 **12.4.3.4.5 Restrictions on generic declarations**

5 This subclause contains the rules that shall be satisfied by every pair of specific procedures that have
 6 the same generic identifier within a scoping unit. If a generic procedure is accessed from a module, the
 7 rules apply to all the specific versions even if some of them are inaccessible by their specific names.

NOTE 12.11

In most scoping units, the possible sources of procedures with a particular generic identifier are the accessible interface blocks and the generic bindings other than names for the accessible objects in that scoping unit. In a type definition, they are the generic bindings, including those from a parent type.

8 A dummy argument is type, kind, and rank compatible, or **TKR compatible**, with another dummy
 9 argument if both have the same rank, and either the first is type compatible with the second and the
 10 kind type parameters of the first have the same values as the corresponding kind type parameters of the
 11 second, or the two are bits compatible.

12 Two dummy arguments are **distinguishable** if

- 13 • one is a procedure and the other is a data object,
- 14 • they are both data objects or known to be functions, and neither is TKR compatible with the
 15 other,
- 16 • one has the ALLOCATABLE attribute and the other has the POINTER attribute, or
- 17 • one is a function with nonzero rank and the other is not known to be a function.

18 Within a scoping unit, if two procedures have the same generic operator and the same number of
 19 arguments or both define assignment, one shall have a dummy argument that corresponds by position
 20 in the argument list to a dummy argument of the other that is distinguishable with it.

1 Within a scoping unit, if two procedures have the same *dtio-generic-spec* (12.4.3.2), their **dtv** arguments
2 shall be type incompatible or have different kind type parameters.

3 Within a scoping unit, two procedures that have the same generic name shall both be subroutines or
4 both be functions, and

- 5 (1) there is a non-passed-object dummy data object in one or the other of them such that
 - 6 (a) the number of dummy data objects in one that are nonoptional, are not passed-object,
7 and with which that dummy data object is TKR compatible, possibly including that
8 dummy data object itself,
9 exceeds
 - 10 (b) the number of non-passed-object dummy data objects, both optional and nonoptional,
11 in the other that are not distinguishable with that dummy data object,
- 12 (2) both have passed-object dummy arguments and the passed-object dummy arguments are
13 distinguishable, or
- 14 (3) at least one of them shall have both
 - 15 (a) a nonoptional non-passed-object dummy argument at an effective position such that
16 either the other procedure has no dummy argument at that effective position or the
17 dummy argument at that position is distinguishable with it, and
 - 18 (b) a nonoptional non-passed-object dummy argument whose name is such that either the
19 other procedure has no dummy argument with that name or the dummy argument
20 with that name is distinguishable with it.
- 21 and the dummy argument that disambiguates by position shall either be the same as or
22 occur earlier in the argument list than the one that disambiguates by name.

23 The **effective position** of a dummy argument is its position in the argument list after any passed-object
24 dummy argument has been removed.

25 Within a scoping unit, if a generic name is the same as the name of a generic intrinsic procedure, the
26 generic intrinsic procedure is not accessible if the procedures in the interface and the intrinsic procedure
27 are not all functions or not all subroutines. If a generic invocation applies to both a specific procedure
28 from an interface and an accessible generic intrinsic procedure, it is the specific procedure from the
29 interface that is referenced.

NOTE 12.12

An extensive explanation of the application of these rules is in C.12.2.

30 12.4.3.5 EXTERNAL statement

31 An **EXTERNAL statement** specifies the EXTERNAL attribute (5.3.8) for a list of names.

32 R1210 *external-stmt* **is** EXTERNAL [::] *external-name-list*

33 The appearance of the name of a block data program unit in an EXTERNAL statement confirms that
34 the block data program unit is a part of the program.

NOTE 12.13

For explanatory information on potential portability problems with external procedures, see sub-
clause C.9.1.

NOTE 12.14

An example of an EXTERNAL statement is:

NOTE 12.14 (cont.)

EXTERNAL FOCUS

1 12.4.3.6 Procedure declaration statement

2 A **procedure declaration statement** declares procedure pointers, dummy procedures, and external
3 procedures. It specifies the EXTERNAL attribute (5.3.8) for all entities in the *proc-decl-list*.

4 R1211 *procedure-declaration-stmt* is PROCEDURE ([*proc-interface*]) ■
5 ■ [[, *proc-attr-spec*] ... ::] *proc-decl-list*

6 R1212 *proc-interface* is *interface-name*
7 or *declaration-type-spec*

8 R1213 *proc-attr-spec* is *access-spec*
9 or *proc-language-binding-spec*
10 or INTENT (*intent-spec*)
11 or OPTIONAL
12 or POINTER
13 or SAVE

14 R1214 *proc-decl* is *procedure-entity-name* [=> *proc-pointer-init*]

15 R1215 *interface-name* is *name*

16 R1216 *proc-pointer-init* is *null-init*
17 or *initial-proc-target*

18 R1217 *initial-proc-target* is *procedure-name*
19

20 C1213 (R1215) The *name* shall be the name of an abstract interface or of a procedure that has an
21 explicit interface. If *name* is declared by a *procedure-declaration-stmt* it shall be previously
22 declared. If *name* denotes an intrinsic procedure it shall be one that is listed in 13.6 and not
23 marked with a bullet (●).

24 C1214 (R1215) The *name* shall not be the same as a keyword that specifies an intrinsic type.

25 C1215 If a procedure entity has the INTENT attribute or SAVE attribute, it shall also have the
26 POINTER attribute.

27 C1216 (R1211) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall
28 specify an external procedure.

29 C1217 (R1214) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.

30 C1218 (R1217) The *procedure-name* shall be the name of an initialization target.

31 C1219 (R1211) If *proc-language-binding-spec* with a NAME= is specified, then *proc-decl-list* shall con-
32 tain exactly one *proc-decl*, which shall neither have the POINTER attribute nor be a dummy
33 procedure.

34 C1220 (R1211) If *proc-language-binding-spec* is specified, the *proc-interface* shall appear, it shall be an
35 *interface-name*, and *interface-name* shall be declared with a *proc-language-binding-spec*.

36 If *proc-interface* appears and consists of *interface-name*, it specifies an explicit specific interface (12.4.3.2)
37 for the declared procedures or procedure pointers. The abstract interface (12.4) is that specified by the

- 1 interface named by *interface-name*.
- 2 If *proc-interface* appears and consists of *declaration-type-spec*, it specifies that the declared procedures
3 or procedure pointers are functions having implicit interfaces and the specified result type. If a type is
4 specified for an external function, its function definition (12.6.2.2) shall specify the same result type and
5 type parameters.
- 6 If *proc-interface* does not appear, the procedure declaration statement does not specify whether the
7 declared procedures or procedure pointers are subroutines or functions.
- 8 If a *proc-attr-spec* other than a *proc-language-binding-spec* appears, it specifies that the declared proce-
9 dures or procedure pointers have that attribute. These attributes are described in 5.3. If a *proc-language-*
10 *binding-spec* with NAME= appears, it specifies a binding label or its absence, as described in 15.5.2.
11 A *proc-language-binding-spec* without a NAME= is allowed, but is redundant with the *proc-interface*
12 required by C1220.
- 13 A procedure is an **initialization target** if it is a nonelemental external or module procedure, or a
14 specific intrinsic function listed in 13.6 and not marked with a bullet (●).
- 15 If => appears in a *proc-decl* in a *procedure-declaration-stmt* it specifies the initial association status
16 of the corresponding procedure entity, and implies the SAVE attribute, which may be confirmed by
17 explicit specification. If => *null-init* appears, the procedure entity is initially disassociated. If =>
18 *initial-proc-target* appears, the procedure entity is initially associated with the target.
- 19 If *proc-entity-name* has an explicit interface, its characteristics shall be the same as *initial-proc-target*
20 except that *initial-proc-target* may be pure even if *proc-entity-name* is not pure and *initial-proc-target*
21 may be an elemental intrinsic procedure.
- 22 If the characteristics of *proc-entity-name* or *initial-proc-target* are such that an explicit interface is
23 required, both *proc-entity-name* and *initial-proc-target* shall have an explicit interface.
- 24 If *proc-entity-name* has an implicit interface and is explicitly typed or referenced as a function, *initial-*
25 *proc-target* shall be a function. If *proc-entity-name* has an implicit interface and is referenced as a
26 subroutine, *initial-proc-target* shall be a subroutine.
- 27 If *initial-proc-target* and *proc-entity-name* are functions, they shall have the same type; corresponding
28 type parameters shall either both be deferred or both have the same value.

NOTE 12.15

In contrast to the EXTERNAL statement, it is not possible to use the PROCEDURE statement to identify a BLOCK DATA subprogram.

NOTE 12.16

The following code illustrates PROCEDURE statements. Note 7.48 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)

```

NOTE 12.16 (cont.)

```

    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 :: 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

```

1 12.4.3.7 INTRINSIC statement

2 An **INTRINSIC statement** specifies the INTRINSIC attribute (5.3.10) for a list of names.

3 R1218 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*

4 C1221 (R1218) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

NOTE 12.17

A name shall not be explicitly specified to have both the EXTERNAL and INTRINSIC attributes in the same scoping unit.

5 12.4.3.8 Implicit interface specification

6 In a scoping unit where the interface of a function is implicit, the type and type parameters of the
 7 function result are specified by an implicit or explicit type specification of the function name. The type,
 8 type parameters, and shape of dummy arguments of a procedure invoked from a scoping unit where the
 9 interface of the procedure is implicit shall be such that the actual arguments are consistent with the
 10 characteristics of the dummy arguments.

11 12.5 Procedure reference**12 12.5.1 Syntax**

13 The form of a procedure reference is dependent on the interface of the procedure or procedure pointer,
 14 but is independent of the means by which the procedure is defined. The forms of procedure references
 15 are as follows.

16 R1219 *function-reference* **is** *procedure-designator* ([*actual-arg-spec-list*])

17 C1222 (R1219) The *procedure-designator* shall designate a function.

- 1 C1223 (R1219) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.
- 2 R1220 *call-stmt* **is** *CALL procedure-designator* [([*actual-arg-spec-list*])]
- 3 C1224 (R1220) The *procedure-designator* shall designate a subroutine.
- 4 R1221 *procedure-designator* **is** *procedure-name*
5 **or** *proc-component-ref*
6 **or** *data-ref* % *binding-name*
- 7 C1225 (R1221) A *procedure-name* shall be the name of a procedure or procedure pointer.
- 8 C1226 (R1221) A *binding-name* shall be a binding name (4.5.5) of the declared type of *data-ref*.
- 9 C1227 (R1221) If *data-ref* is an array, the referenced type-bound procedure shall have the PASS at-
10 tribute.
- 11 Resolving references to type-bound procedures is described in 12.5.6.
- 12 A function may also be referenced as a defined operation (12.4.3.4.2). A subroutine may also be ref-
13 erenced as a defined assignment (12.4.3.4.3), by user-defined derived-type input/output (9.5.4.7), or by
14 finalization (4.5.6).

NOTE 12.18

If image *I* executes a procedure reference in which the *variable* of a *proc-component-ref* specifies a procedure pointer on image *J*, the procedure pointer association is fetched from image *J* but the invocation of the associated procedure occurs on image *I*.

- 15 R1222 *actual-arg-spec* **is** [*keyword* =] *actual-arg*
- 16 R1223 *actual-arg* **is** *expr*
17 **or** *variable*
18 **or** *procedure-name*
19 **or** *proc-component-ref*
20 **or** *alt-return-spec*
- 21 R1224 *alt-return-spec* **is** * *label*
- 22 C1228 (R1222) The *keyword* = shall not appear if the interface of the procedure is implicit in the
23 scoping unit.
- 24 C1229 (R1222) The *keyword* = shall not be omitted from an *actual-arg-spec* unless it has been omitted
25 from each preceding *actual-arg-spec* in the argument list.
- 26 C1230 (R1222) Each *keyword* shall be the name of a dummy argument in the explicit interface of the
27 procedure.
- 28 C1231 (R1223) A nonintrinsic elemental procedure shall not be used as an actual argument.
- 29 C1232 (R1223) A *procedure-name* shall be the name of an external, internal, module, or dummy proce-
30 dure, a specific intrinsic function listed in 13.6 and not marked with a bullet (•), or a procedure
31 pointer.
- 32 C1233 (R1224) The *label* shall be the statement label of a branch target statement that appears in the same scoping
33 unit as the *call-stmt*.

NOTE 12.19

Successive commas shall not be used to omit optional arguments.

NOTE 12.20

Examples of procedure reference using procedure pointers:

```
P => BESSEL
WRITE (*, *) P(2.5)      !-- BESSEL(2.5)

S => PRINT_REAL
CALL S(3.14)
```

NOTE 12.21

An internal procedure cannot be invoked using a procedure pointer from either Fortran or C after the host instance completes execution, because the pointer is then undefined. While the host instance is active, however, the internal procedure may be invoked from outside of the host procedure scoping unit if that internal procedure was passed as an actual argument or is the target of a procedure pointer.

Let us assume there is a procedure with the following interface that calculates $\int_a^b f(x) dx$.

```
INTERFACE
  FUNCTION INTEGRATE(F, A, B) RESULT(INTEGRAL) BIND(C)
    USE ISO_C_BINDING
    INTERFACE
      FUNCTION F(X) BIND(C) ! Integrand
        USE ISO_C_BINDING
        REAL(C_FLOAT), VALUE :: X
        REAL(C_FLOAT) :: F
      END FUNCTION
    END INTERFACE
    REAL(C_FLOAT), VALUE :: A, B ! Bounds
    REAL(C_FLOAT) :: INTEGRAL
  END FUNCTION INTEGRATE
END INTERFACE
```

This procedure can be called from Fortran or C, and could be written in either Fortran or C. The argument F representing the mathematical function $f(x)$ can be written as an internal procedure; this internal procedure will have access to any host instance local variables necessary to actually calculate $f(x)$. For example:

```
REAL FUNCTION MY_INTEGRATION(N, A, B) RESULT(INTEGRAL)
  ! Integrate f(x)=x^n over [a,b]
  USE ISO_C_BINDING
  INTEGER, INTENT(IN) :: N
  REAL, INTENT(IN) :: A, B

  INTEGRAL = INTEGRATE(MY_F, REAL(A, C_FLOAT), REAL(B, C_FLOAT))
  ! This will call the internal function MY_F to calculate f(x).
  ! The above interface of INTEGRATE must be explicit and available.
```

CONTAINS

NOTE 12.21 (cont.)

```

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 shall not save 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 saved, then it can only be used to invoke MY_F during the execution of the host instance of MY_INTEGRATION called from INTEGRATE.

1 12.5.2 Actual arguments, dummy arguments, and argument association

2 12.5.2.1 Argument correspondence

3 In either a subroutine reference or a function reference, the actual argument list identifies the corre-
 4 spondence between the actual arguments supplied and the dummy arguments of the procedure. This
 5 correspondence may be established either by keyword or by position. If an argument keyword appears,
 6 the actual argument corresponds to the dummy argument whose name is the same as the argument
 7 keyword (using the dummy argument names from the interface accessible in the scoping unit containing
 8 the procedure reference). In the absence of an argument keyword, an actual argument corresponds to the
 9 dummy argument occupying the corresponding position in the reduced dummy argument list; that is,
 10 the first actual argument corresponds to the first dummy argument in the reduced list, the second actual
 11 argument corresponds to the second dummy argument in the reduced list, etc. The reduced dummy
 12 argument list is either the full dummy argument list or, if there is a passed-object dummy argument
 13 (4.5.4.4), the dummy argument list with the passed object dummy argument omitted. Exactly one
 14 actual argument shall correspond to each nonoptional dummy argument. At most one actual argument
 15 shall correspond to each optional dummy argument. Each actual argument shall correspond to a dummy
 16 argument.

NOTE 12.22

For example, the procedure defined by

```

SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
  INTERFACE
    FUNCTION FUNCT (X)
      REAL FUNCT, X
    END FUNCTION FUNCT
  END INTERFACE
  REAL SOLUTION
  INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
  ...

```

may be invoked with

```

CALL SOLVE (FUN, SOL, PRINT = 6)

```

provided its interface is explicit; if the interface is specified by an interface block, the name of the last argument shall be PRINT.

1 12.5.2.2 The passed-object dummy argument and argument correspondence

2 In a reference to a type-bound procedure, or a procedure pointer component, that has a (4.5.4.4), the
3 *data-ref* of the *function-reference* or *call-stmt* corresponds, as an actual argument, with the passed-object
4 dummy argument.

5 12.5.2.3 Argument association

6 Except in references to intrinsic inquiry functions, a pointer actual argument that corresponds to a
7 nonoptional nonpointer dummy argument shall be pointer associated with a target.

8 If a nonpointer dummy argument without the VALUE attribute corresponds to a pointer actual argument
9 that is pointer associated with a target, the dummy argument becomes argument associated with that
10 target.

11 If a present nonpointer dummy argument without the VALUE attribute corresponds to a nonpointer
12 actual argument it becomes argument associated with that actual argument.

13 A present dummy argument with the VALUE attribute becomes argument associated with a definable
14 anonymous data object whose initial value is the value of the actual argument.

15 A present pointer dummy argument that corresponds to a pointer actual argument becomes argument
16 associated with that actual argument. A present pointer dummy argument that does not correspond to
17 a pointer actual argument is not argument associated.

18 The entity that is argument-associated with a dummy argument is called its **effective argument**.

19 12.5.2.4 Compatibility with bits objects

20 An entity is **bits compatible** with another entity if and only if one is of type bits, the other is of type
21 bits, integer, real, complex, or logical, and scalar entities of these types have the same size expressed in
22 bits.

J3 internal note

Unresolved Technical Issue 016

Can “bits compatible” be folded into a new concept of “type and kind compatible”?

23 12.5.2.5 Ordinary dummy variables

24 The requirements in this subclause apply to actual arguments that correspond to nonallocatable non-
25 pointer dummy data objects.

26 The dummy argument shall be type compatible with the actual argument or it shall be of type bits and
27 bits compatible with the actual argument.

28 Unless the actual argument and the corresponding dummy argument are bits compatible, the type
29 parameter values of the actual argument shall agree with the corresponding ones of the dummy argument
30 that are not assumed, except for the case of the character length parameter of an actual argument of
31 type default character associated with a dummy argument that is not assumed shape.

32 If a scalar dummy argument is of type default character, the length *len* of the dummy argument shall
33 be less than or equal to the length of the actual argument. The dummy argument becomes associated
34 with the leftmost *len* characters of the actual argument. If an array dummy argument is of type default
35 character and is not assumed shape, it becomes associated with the leftmost characters of the actual
36 argument element sequence (12.5.2.12).

37 The values of assumed type parameters of a dummy argument are assumed from the corresponding type

- 1 parameters of the actual argument.
- 2 If the actual argument is a co-indexed object with an allocatable ultimate component, the dummy
- 3 argument shall have the INTENT(IN) or the VALUE attribute.

J3 internal note

Unresolved Technical Issue 050

Note that this does not match the edit in 06-319r3, but I believe it correctly reflects the technical discussion therein.

Remove this UTI if that is correct, and if the requirement is good.

NOTE 12.23

If the actual argument is a co-indexed 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 subcomponents, 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.

- 4 Except in references to intrinsic inquiry functions, if the dummy argument is nonoptional and the actual
- 5 argument is allocatable, the corresponding actual argument shall be allocated.

NOTE 12.24

Fortran argument association is usually similar to call by reference and call by value-result. If the VALUE attribute is specified, the effect is as if the actual argument is assigned to a temporary, and the temporary is then argument associated with the dummy argument. Subsequent changes to the value or definition status of the dummy argument do not affect the actual argument. The actual mechanism by which this happens is determined by the processor.

- 6 If the dummy argument does not have the TARGET attribute, or is not type-compatible with the
- 7 actual argument, any pointers associated with the effective argument do not become associated with the
- 8 corresponding dummy argument on invocation of the procedure. If such a dummy argument is used as
- 9 an actual argument that corresponds to a dummy argument with the TARGET attribute, whether any
- 10 pointers associated with the original effective argument become associated with the dummy argument
- 11 with the TARGET attribute is processor dependent.
- 12 If the dummy argument has the TARGET attribute, is type-compatible with the actual argument, does
- 13 not have the VALUE attribute, and is either a scalar or an assumed-shape array that does not have the
- 14 CONTIGUOUS attribute, and the effective argument has the TARGET attribute but is not a co-indexed
- 15 object or an array section with a vector subscript then
- 16 (1) any pointers associated with the effective argument become associated with the correspond-
- 17 ing dummy argument on invocation of the procedure, and
- 18 (2) when execution of the procedure completes, any pointers that do not become undefined
- 19 (16.5.2.2.3) and are associated with the dummy argument remain associated with the effec-
- 20 tive argument.
- 21 If the dummy argument has the TARGET attribute and is an explicit-shape array, an assumed-shape
- 22 array with the CONTIGUOUS attribute, or an assumed-size array, and the effective argument has the
- 23 TARGET attribute but is not an array section with a vector subscript then
- 24 (1) on invocation of the procedure, whether any pointers associated with the effective argument
- 25 become associated with the corresponding dummy argument is processor dependent, and
- 26 (2) when execution of the procedure completes, the pointer association status of any pointer
- 27 that is pointer associated with the dummy argument is processor dependent.
- 28 If the dummy argument has the TARGET attribute and the effective argument does not have the

- 1 TARGET attribute or is an array section with a vector subscript, or the dummy argument is not
 2 type-compatible with the actual argument, any pointers associated with the dummy argument become
 3 undefined when execution of the procedure completes.
- 4 If the dummy argument has the TARGET attribute and the VALUE attribute, any pointers associated
 5 with the dummy argument become undefined when execution of the procedure completes.
- 6 If the actual argument is scalar, the corresponding dummy argument shall be scalar unless the actual
 7 argument is of type default character, of type character with the C character kind (15.2), or is an element
 8 or substring of an element of an array that is not an assumed-shape, pointer, or polymorphic array. If
 9 the procedure is nonelemental and is referenced by a generic name or as a defined operator or defined
 10 assignment, the ranks of the actual arguments and corresponding dummy arguments shall agree.
- 11 If a dummy argument is an assumed-shape array, the rank of the actual argument shall be the same as
 12 the rank of the dummy argument; the actual argument shall not be an assumed-size array (including an
 13 array element designator or an array element substring designator).
- 14 Except when a procedure reference is elemental (12.8), each element of an array actual argument or of
 15 a sequence in a sequence association (12.5.2.12) is associated with the element of the dummy array that
 16 has the same position in array element order (6.2.2.1).

NOTE 12.25

For type default character sequence associations, the interpretation of element is provided in 12.5.2.12.

- 17 A scalar dummy argument of a nonelemental procedure shall be associated only with a scalar actual
 18 argument.
- 19 If a dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument shall be definable.
 20 If a dummy argument has INTENT (OUT), the actual argument becomes undefined at the time the
 21 association is established, except for direct components of an object of derived type for which default
 22 initialization has been specified. If the dummy argument is not polymorphic and the type of the effective
 23 argument is an extension type of the type of the dummy argument, only the part of the effective argument
 24 that is of the same type as the dummy argument becomes undefined.
- 25 If the actual argument is an array section having a vector subscript, the dummy argument is not defin-
 26 able and shall not have the ASYNCHRONOUS, INTENT (OUT), INTENT (INOUT), or VOLATILE
 27 attributes.

NOTE 12.26

Argument intent specifications serve several purposes. See Note 5.16.

NOTE 12.27

For more explanatory information on argument association and evaluation, see subclause C.9.5. For more explanatory information on targets as dummy arguments, see subclause C.9.6.

- 28 C1234 An actual argument that is a co-indexed object shall not correspond to a dummy argument that
 29 has either the ASYNCHRONOUS or VOLATILE attribute.
- 30 C1235 (R1223) If an actual argument is an array section or an assumed-shape array, and the corre-
 31 sponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that
 32 dummy argument shall be an assumed-shape array that does not have the CONTIGUOUS
 33 attribute.

- 1 C1236 (R1223) If an actual argument is a pointer array, and the corresponding dummy argument
 2 has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an
 3 assumed-shape array that does not have the CONTIGUOUS attribute or a pointer array.

J3 internal note

Unresolved Technical Issue 097

So, consider this example.

```

INTERFACE ASSIGNMENT(=)
  SUBROUTINE s(a,b)
    REAL,INTENT(OUT),VOLATILE :: a(*)
    REAL,INTENT(IN) :: b(:)
  END SUBROUTINE
END
REAL,POINTER :: p(:),q(:)
...
CALL s(p,q) ! Constraint violation associating P with A
p = q      ! No constraint violation because <actual-arg>
           ! syntax is not being used.

```

Hmm, maybe this should be an interp?

Is there any doubt as to our intention?

Is there any doubt the standard does not reflect that intention?

NOTE 12.28

The constraints on actual arguments that correspond 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 actual arguments whose values are likely to change due to an asynchronous I/O operation completing or in some unpredictable manner will cause those new values to be lost when a called procedure returns and the copy-out overwrites the actual argument.

4 12.5.2.6 Allocatable and pointer dummy variables

- 5 The requirements in this subclause apply to actual arguments that correspond to either allocatable or
 6 pointer dummy data objects.
- 7 The actual argument shall be polymorphic if and only if the associated dummy argument is polymorphic,
 8 and either both the actual and dummy arguments shall be unlimited polymorphic, or the declared type
 9 of the actual argument shall be the same as the declared type of the dummy argument.

NOTE 12.29

The dynamic type of a polymorphic allocatable or pointer dummy argument may change as a result of execution of an allocate statement or pointer assignment in the subprogram. Because of this the corresponding actual argument needs to be polymorphic and have a declared type that is the same as the declared type of the dummy argument or an extension of that type. However, type compatibility requires that the declared type of the dummy argument be the same as, or an extension of, the type of the actual argument. Therefore, the dummy and actual arguments need to have the same declared type.

Dynamic type information is not maintained for a nonpolymorphic allocatable or pointer dummy argument. However, allocating or pointer assigning such a dummy argument would require maintenance of this information if the corresponding actual argument is polymorphic. Therefore, the corresponding actual argument needs to be nonpolymorphic.

- 1 The rank of the actual argument shall be the same as that of the dummy argument. The type parameter
- 2 values of the actual argument shall agree with the corresponding ones of the dummy argument that are
- 3 not assumed or deferred.
- 4 The values of assumed type parameters of a dummy argument are assumed from the corresponding type
- 5 parameters of the associated actual argument.
- 6 The actual argument shall have deferred the same type parameters as the dummy argument.
- 7 If the actual argument is a co-indexed object, the dummy argument shall have the INTENT(IN) at-
- 8 tribute.

J3 internal note

Unresolved Technical Issue 096

Co-indexed actual to pointer/allocatable dummy.

In the interests of preventing remote allocation/deallocation or remote pointer assignment, in accordance with the reasoning in 06-319r3, it would seem necessary to have the above requirement specified somewhere. This seems like the obvious place to have it: this subclause contains those requirements which apply both to allocatable dummies and to pointer dummies.

It's possible that the requirement already exists somewhere else, in which case it's superfluous here ... but the authors of 06-319r3 clearly didn't think so, at least in the allocatable case.

If this requirement is too onerous for the pointer case, just move the above sentence to the *next* subclause, which applies only to the allocatable case.

Note that this might have some "consistency" interaction with what is left of UTI 050.

9 12.5.2.7 Allocatable dummy variables

- 10 The requirements in this subclause apply to actual arguments that correspond to allocatable dummy
- 11 data objects.
- 12 The actual argument shall be allocatable. It is permissible for the actual argument to have an allocation
- 13 status of unallocated.
- 14 If the dummy argument does not have the TARGET attribute, any pointers associated with the ac-
- 15 tual argument do not become associated with the corresponding dummy argument on invocation of the
- 16 procedure. If such a dummy argument is used as an actual argument that is associated with a dummy ar-
- 17 gument with the TARGET attribute, whether any pointers associated with the original actual argument
- 18 become associated with the dummy argument with the TARGET attribute is processor dependent.
- 19 If the dummy argument has the TARGET attribute, does not have the INTENT(OUT) or VALUE
- 20 attribute, and the corresponding actual argument has the TARGET attribute then
 - 21 (1) any pointers associated with the actual argument become associated with the corresponding
 - 22 dummy argument on invocation of the procedure, and
 - 23 (2) when execution of the procedure completes, any pointers that do not become undefined
 - 24 (16.5.2.2.3) and are associated with the dummy argument remain associated with the actual
 - 25 argument.
- 26 If a dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument shall be definable.
- 27 If a dummy argument has INTENT (OUT), an allocated actual argument is deallocated on procedure
- 28 invocation (6.3.3.1).

29 12.5.2.8 Pointer dummy variables

- 30 The requirements in this subclause apply to actual arguments that correspond to dummy data pointers.
- 31 If the dummy argument does not have the INTENT(IN) attribute, the actual argument shall be a

1 pointer. Otherwise, the actual argument shall be a pointer or a valid target for the dummy pointer in
 2 a pointer assignment statement. If the actual argument is not a pointer, the dummy pointer becomes
 3 pointer-associated with the actual argument.

4 The nondeferred type parameters and ranks shall agree.

5 If the dummy pointer has the CONTIGUOUS attribute and the actual argument does not have the
 6 CONTIGUOUS attribute, the actual argument shall satisfy the following conditions.

- 7 • Its base object shall either have the CONTIGUOUS attribute or shall not be a pointer or assumed-
 8 shape array.
- 9 • It shall not be the real or imaginary part of an array of type complex.
- 10 • Its designator shall not contain a *substring-range*.
- 11 • It shall not have a vector subscript.
- 12 • Only its final *part-ref* shall have nonzero rank.
- 13 • If that *part-ref* has a *section-subscript-list*, the *section-subscript-list* shall satisfy these conditions:
 14 – no *stride* shall appear;
 15 – if any *section-subscript* is a subscript, it shall not be followed by a *subscript-triplet*;
 16 – all but the last *subscript-triplet* shall consist of a single colon with no *subscript*.

NOTE 12.30

This requires that the actual argument be obviously contiguous at compile time. Columns, planes, cubes and hypercubes of contiguous base objects can be passed, for example:

```

ARRAY1 (10:20, 3)    ! passes part of the third column of ARRAY1.
X3D (:, i:j, 2)      ! passes part of the second plane of X3D (or the whole
                     ! plane if i==LBOUND(X3D,2) and j==UBOUND(X3D,2).
Y5D (:, :, :, :, 7) ! passes the seventh hypercube of Y5D.
```

17 If the dummy argument has INTENT(OUT), the pointer association status of the actual argument
 18 becomes undefined on invocation of the procedure.

19 If the dummy argument is nonoptional and the actual argument is allocatable, the actual argument shall
 20 be allocated.

NOTE 12.31

For more explanatory information on pointers as dummy arguments, see subclause C.9.6.

21 12.5.2.9 Co-array arguments

22 If the dummy argument is an allocatable co-array, the actual argument shall be an allocatable co-array
 23 with the same rank and co-rank.

24 If a dummy argument is a nonallocatable co-array, the co-rank and co-bounds are specified by the local
 25 declaration and are independent of those of the actual argument. The actual argument shall be a co-array
 26 or a subobject of a co-array without any co-subscripts, vector-valued subscripts, non-co-array allocatable
 27 component selection, or pointer component selection.

NOTE 12.32

The co-array on the executing image is specified as the actual argument and this is associated with the dummy co-array argument on the same image. There are corresponding co-arrays on the other images. For example:

```

INTERFACE
  SUBROUTINE SUB(X, Y)
    REAL :: X(:)[*], Y(:)[*]
  END SUBROUTINE SUB
END INTERFACE
...
REAL, ALLOCATABLE :: A(:)[:], B(:,:)[:]
...
CALL SUB(A(:), B(1,:))

```

NOTE 12.33

The bounds and co-bounds of a dummy co-array may differ on each image executing a reference to the procedure. A co-indexed *data-ref* always uses the bounds and co-bounds on the executing image.

- 1 If the dummy argument is a co-array that has the CONTIGUOUS attribute or is not of assumed shape,
- 2 the actual argument shall be compile-time contiguous.

J3 internal note

Unresolved Technical Issue 101

Use of undefined term “compile-time contiguous”.

This term has not been defined. According to a straw vote taken at meeting 178, it ought to be.

NOTE 12.34

The requirements on an actual argument that corresponds to a dummy co-array 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.

3 12.5.2.10 Actual arguments associated with dummy procedure entities

- 4 If the actual argument is the name of an internal subprogram, the host instance of the dummy argument
- 5 is the innermost currently executing instance of the host of that internal subprogram. If the actual
- 6 argument has a host instance the host instance of the dummy argument is that instance. Otherwise the
- 7 dummy argument has no host instance.
- 8 If a dummy argument is a procedure pointer, the associated actual argument shall be a procedure pointer,
- 9 a reference to a function that returns a procedure pointer, a reference to the NULL intrinsic function, or
- 10 a valid target for the dummy pointer in a pointer assignment statement. If the actual argument is not
- 11 a pointer, the dummy argument shall have the INTENT(IN) attribute and becomes pointer associated
- 12 with the actual argument.
- 13 If a dummy argument is a dummy procedure without the POINTER attribute, the associated actual
- 14 argument shall be an external, internal, module, or dummy procedure, or a specific intrinsic procedure
- 15 listed in 13.6 and not marked with a bullet (●). If the specific name is also a generic name, only the
- 16 specific procedure is associated with the dummy argument.
- 17 If an external procedure name or a dummy procedure name is used as an actual argument, its interface

- 1 shall be explicit or it shall be explicitly declared to have the EXTERNAL attribute.
- 2 If the interface of a dummy procedure is explicit, the characteristics listed in 12.3.1 shall be the same
 3 for the associated actual argument and the corresponding dummy argument, except that a pure actual
 4 argument may be associated with a dummy argument that is not pure and an elemental intrinsic actual
 5 procedure may be associated with a dummy procedure (which is prohibited from being elemental).
- 6 If the interface of a dummy procedure is implicit and either the dummy argument is explicitly typed
 7 or referenced as a function, it shall not be referenced as a subroutine and any corresponding actual
 8 argument shall be a function, function procedure pointer, or dummy procedure.
- 9 If the interface of a dummy procedure is implicit and a reference to it appears as a subroutine reference,
 10 any corresponding actual argument shall be a subroutine, subroutine procedure pointer, or dummy
 11 procedure.

12 12.5.2.11 Actual arguments associated with alternate return indicators

- 13 If a dummy argument is an asterisk (12.6.2.3), the associated actual argument shall be an alternate return specifier (12.5).

14 12.5.2.12 Sequence association

15 An actual argument represents an **element sequence** if it is an array expression, an array element
 16 designator, a scalar of type default character, or a scalar of type character with the C character kind
 17 (15.2.2). If the actual argument is an array expression, the element sequence consists of the elements
 18 in array element order. If the actual argument is an array element designator, the element sequence
 19 consists of that array element and each element that follows it in array element order.

20 If the actual argument is of type default character or of type character with the C character kind, and is
 21 an array expression, array element, or array element substring designator, the element sequence consists
 22 of the storage units beginning with the first storage unit of the actual argument and continuing to the
 23 end of the array. The storage units of an array element substring designator are viewed as array elements
 24 consisting of consecutive groups of storage units having the character length of the dummy array.

25 If the actual argument is of type default character or of type character with the C character kind, and
 26 is a scalar that is not an array element or array element substring designator, the element sequence
 27 consists of the storage units of the actual argument.

NOTE 12.35

Some of the elements in the element sequence may consist of storage units from different elements of the original array.

28 An actual argument that represents an element sequence and corresponds to a dummy argument that is
 29 an array is sequence associated with the dummy argument if the dummy argument is an explicit-shape
 30 or assumed-size array. The rank and shape of the actual argument need not agree with the rank and
 31 shape of the dummy argument, but the number of elements in the dummy argument shall not exceed
 32 the number of elements in the element sequence of the actual argument. If the dummy argument is
 33 assumed-size, the number of elements in the dummy argument is exactly the number of elements in the
 34 element sequence.

35 12.5.2.13 Argument presence and restrictions on arguments not present

36 A dummy argument or an entity that is host associated with a dummy argument is not **present** if the
 37 dummy argument

- 38 (1) does not correspond to an actual argument,

- 1 (2) corresponds to an actual argument that is not present, or
- 2 (3) does not have the ALLOCATABLE or POINTER attribute, and corresponds to an actual
- 3 argument that
- 4 (a) has the ALLOCATABLE attribute and is not allocated, or
- 5 (b) has the POINTER attribute and is disassociated.

6 Otherwise, it is present. A nonoptional dummy argument shall be present. If an optional nonpointer
 7 dummy argument corresponds to a pointer actual argument, the pointer association status of the actual
 8 argument shall not be undefined.

9 An optional dummy argument that is not present is subject to the following restrictions.

- 10 (1) If it is a data object, it shall not be referenced or be defined. If it is of a type for which
- 11 default initialization is specified for a direct component, the initialization has no effect.
- 12 (2) It shall not be used as the *data-target* or *proc-target* of a pointer assignment.
- 13 (3) If it is a procedure or procedure pointer, it shall not be invoked.
- 14 (4) It shall not be supplied as an actual argument corresponding to a nonoptional dummy
- 15 argument other than as the argument of the PRESENT intrinsic function or as an argument
- 16 of a function reference that meets the requirements of (6) or (8) in 7.1.12.
- 17 (5) A designator with it as the base object and with at least one subobject selector shall not
- 18 be supplied as an actual argument.
- 19 (6) If it is an array, it shall not be supplied as an actual argument to an elemental procedure
- 20 unless an array of the same rank is supplied as an actual argument corresponding to a
- 21 nonoptional dummy argument of that elemental procedure.
- 22 (7) If it is a pointer, it shall not be allocated, deallocated, nullified, pointer-assigned, or supplied
- 23 as an actual argument corresponding to an optional nonpointer dummy argument.
- 24 (8) If it is allocatable, it shall not be allocated, deallocated, or supplied as an actual argument
- 25 corresponding to an optional nonallocatable dummy argument.
- 26 (9) If it has length type parameters, they shall not be the subject of an inquiry.
- 27 (10) It shall not be used as the *selector* in a SELECT TYPE or ASSOCIATE construct.

28 Except as noted in the list above, it may be supplied as an actual argument corresponding to an optional
 29 dummy argument, which is then also considered not to be present.

30 **12.5.2.14 Restrictions on entities associated with dummy arguments**

31 While an entity is associated with a dummy argument, the following restrictions hold.

- 32 (1) Action that affects the allocation status of the entity or a subobject thereof shall be taken
- 33 through the dummy argument. Action that affects the value of the entity or any subobject
- 34 of it shall be taken only through the dummy argument unless
- 35 (a) the dummy argument has the POINTER attribute or
- 36 (b) the dummy argument has the TARGET attribute, the dummy argument does not
- 37 have INTENT (IN), the dummy argument is a scalar object or an assumed-shape
- 38 array without the CONTIGUOUS attribute, and the actual argument is a target
- 39 other than an array section with a vector subscript.

NOTE 12.36

In

```
SUBROUTINE OUTER
  REAL, POINTER :: A (:)
  ...
```

NOTE 12.36 (cont.)

```

    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 12.37

If there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure and the dummy arguments have neither the `POINTER`

NOTE 12.37 (cont.)

nor TARGET attribute, the overlapped portions shall not be defined, redefined, or become undefined during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

A (3:5) shall not be defined, redefined, or become undefined through the first dummy argument because it is part of the argument associated with the second dummy argument and shall not be defined, redefined, or become undefined through the second dummy argument because it is part of the argument associated with the first dummy argument. A (1:2) remains definable through the first dummy argument and A (6:9) remains definable through the second dummy argument.

NOTE 12.38

This restriction applies equally to pointer targets. In

```
REAL, DIMENSION (10), TARGET :: A
REAL, DIMENSION (:), POINTER :: B, C
B => A (1:5)
C => A (3:9)
CALL SUB (B, C) ! The dummy arguments of SUB are neither pointers nor targets.
```

B (3:5) cannot be defined because it is part of the argument associated with the second dummy argument. C (1:3) cannot be defined because it is part of the argument associated with the first dummy argument. A (1:2) [which is B (1:2)] remains definable through the first dummy argument and A (6:9) [which is C (4:7)] remains definable through the second dummy argument.

NOTE 12.39

Because a nonpointer dummy argument declared with INTENT(IN) shall not be used to change the associated actual argument, the associated actual argument remains constant throughout the execution of the procedure.

- 1 (2) If the allocation status of the entity or a subobject thereof is affected through the dummy
- 2 argument, then at any time during the execution of the procedure, either before or after the
- 3 allocation or deallocation, it may be referenced only through the dummy argument. If the
- 4 value of the entity or any subobject of it is affected through the dummy argument, then at
- 5 any time during the execution of the procedure, either before or after the definition, it may
- 6 be referenced only through that dummy argument unless
- 7 (a) the dummy argument has the POINTER attribute or
- 8 (b) the dummy argument has the TARGET attribute, the dummy argument does not
- 9 have INTENT (IN), the dummy argument is a scalar object or an assumed-shape
- 10 array without the CONTIGUOUS attribute, and the actual argument is a target
- 11 other than an array section with a vector subscript.

NOTE 12.40

In

```
MODULE DATA
  REAL :: W, X, Y, Z
END MODULE DATA
```

```
PROGRAM MAIN
```

NOTE 12.40 (cont.)

```

USE DATA
...
CALL INIT (X)
...
END PROGRAM MAIN
SUBROUTINE INIT (V)
  USE DATA
  ...
  READ (*, *) V
  ...
END SUBROUTINE INIT

```

variable X shall not be directly referenced at any time during the execution of INIT because it is being defined through the dummy argument V. X may be (indirectly) referenced through V. W, Y, and Z may be directly referenced. X may, of course, be directly referenced once execution of INIT is complete.

NOTE 12.41

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.

1 12.5.3 Function reference

2 A function is invoked during expression evaluation by a *function-reference* or by a defined operation
 3 (7.1.6). When it is invoked, all actual argument expressions are evaluated, then the arguments are
 4 associated, and then the function is executed. When execution of the function is complete, the value
 5 of the function result is available for use in the expression that caused the function to be invoked. The
 6 characteristics of the function result (12.3.3) are determined by the interface of the function. A reference
 7 to an elemental function (12.8) is an elemental reference if one or more actual arguments are arrays and
 8 all array arguments have the same shape.

9 12.5.4 Subroutine reference

10 A subroutine is invoked by execution of a CALL statement, execution of a defined assignment statement
 11 (7.2.1.4), user-defined derived-type input/output(9.5.4.7.1), or finalization(4.5.6). When a subroutine is
 12 invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the
 13 subroutine is executed. When the actions specified by the subroutine are completed, the execution of the
 14 CALL statement, the execution of the defined assignment statement, the processing of an input or output
 15 list item, or finalization of an object is also completed. If a CALL statement includes one or more alternate
 16 return specifiers among its arguments, control may be transferred to one of the statements indicated, depending on the
 17 action specified by the subroutine. A reference to an elemental subroutine (12.8) is an elemental reference if
 18 there is at least one actual argument corresponding to an INTENT(OUT) or INTENT(INOUT) dummy
 19 argument, all such actual arguments are arrays, and all actual arguments are conformable.

20 12.5.5 Resolving named procedure references

12.5.5.1 Establishment of procedure names

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.

A procedure name is established to be generic in a scoping unit

- (1) if that scoping unit contains an interface block with that name;
- (2) if that scoping unit contains an INTRINSIC attribute specification for that name and it is the name of a generic intrinsic procedure;
- (3) if that scoping unit contains a USE statement that makes that procedure name accessible and the corresponding name in the module is established to be generic; or
- (4) if that scoping unit contains no declarations of that name, that scoping unit has a host scoping unit, and that name is established to be generic in the host scoping unit.

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 that defines a procedure with that name;
- (2) if that scoping unit contains an INTRINSIC attribute specification for that name and it is the name of a specific intrinsic procedure;
- (3) if that scoping unit contains an explicit EXTERNAL attribute specification (5.3.8) for that name;
- (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 specific; 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 specific in the host scoping unit.

A procedure name is not established in a scoping unit if it is neither established to be generic nor established to be specific.

12.5.5.2 Resolving procedure references to names established to be generic

If the reference is consistent with a nonelemental reference to one of the specific interfaces of a generic interface that has that name and either is in the scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific procedure in the interface block that provides that interface. The rules in 12.4.3.4.5 ensure that there can be at most one such specific procedure.

Otherwise, if the reference is consistent with an elemental reference to one of the specific interfaces of a generic interface that has that name and either is in the scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific elemental procedure in the interface block that provides that interface. The rules in 12.4.3.4.5 ensure that there can be at most one such specific elemental procedure.

Otherwise, if the scoping unit contains either an INTRINSIC attribute specification for that name or a USE statement that makes that name accessible from a module in which the corresponding name is specified to have the INTRINSIC attribute, and if the reference is consistent with the interface of that intrinsic procedure, the reference is to that intrinsic procedure.

Otherwise, if the scoping unit has a host scoping unit, the name is established to be generic in that host scoping unit, and there is agreement between the scoping unit and the host scoping unit as to whether

- 1 the name is a function name or a subroutine name, the name is resolved by applying the rules in this
- 2 subclause to the host scoping unit.
- 3 Otherwise, if the name is that of an intrinsic procedure and the reference is consistent with that intrinsic
- 4 procedure, the reference is to that intrinsic procedure.

NOTE 12.42

These rules allow particular specific procedures of a generic procedure 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.

NOTE 12.43

In the USE statement case, it is possible, because of the renaming facility, for the name in the reference to be different from the name of the intrinsic procedure.

5 12.5.5.3 Resolving procedure references to names established to be only specific

- 6 If the scoping unit contains an interface body or EXTERNAL attribute specification for the name and
- 7 the name is the name of a dummy argument of the scoping unit, the dummy argument is a dummy
- 8 procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing
- 9 that reference is the procedure supplied as the actual argument corresponding to that dummy procedure.
- 10 If the scoping unit contains an interface body or EXTERNAL attribute specification for the name and
- 11 the name is not the name of a dummy argument of the scoping unit, the reference is to an external
- 12 procedure with that name.
- 13 If the scoping unit contains a module subprogram, internal subprogram, or statement function that defines
- 14 a procedure with the name, the reference is to the procedure so defined.
- 15 If the scoping unit contains an INTRINSIC attribute specification for the name, the reference is to the

- 1 intrinsic with that name.
- 2 If the scoping unit contains a USE statement that makes a procedure accessible by the name, the
- 3 reference is to that procedure.

NOTE 12.44

Because of the renaming facility of the USE statement, the name in the reference may be different from the original name of the procedure.

- 4 If none of the above apply, the scoping unit shall have a host scoping unit, and the reference is resolved
- 5 by applying the rules in this subclause to the host scoping unit.

6 12.5.5.4 Resolving procedure references to names not established

- 7 If the name is the name of a dummy argument of the scoping unit, the dummy argument is a dummy
- 8 procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing
- 9 that reference is the procedure supplied as the actual argument corresponding to that dummy procedure.

- 10 Otherwise, if the name is the name of an intrinsic procedure, and if there is agreement between the
- 11 reference and the status of the intrinsic procedure as being a function or subroutine, the reference is to
- 12 that intrinsic procedure.

- 13 Otherwise, the reference is to an external procedure with that name.

14 12.5.6 Resolving type-bound procedure references

- 15 If the *binding-name* in a *procedure-designator* (R1221) is that of a specific type-bound procedure, the
- 16 procedure referenced is the one bound to that name in the dynamic type of the *data-ref*.

- 17 If the *binding-name* in a *procedure-designator* is that of a generic type bound procedure, the generic
- 18 binding with that name in the declared type of the *data-ref* is used to select a specific binding using the
- 19 following criteria.

- 20 (1) If the reference is consistent with one of the specific bindings of that generic binding, that
- 21 specific binding is selected.
- 22 (2) Otherwise, the reference shall be consistent with an elemental reference to one of the specific
- 23 bindings of that generic binding; that specific binding is selected.

- 24 The reference is to the procedure bound to the same name as the selected specific binding in the dynamic
- 25 type of the *data-ref*.

26 12.6 Procedure definition**27 12.6.1 Intrinsic procedure definition**

- 28 Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor
- 29 shall include the intrinsic procedures described in Clause 13, but may include others. However, a
- 30 standard-conforming program shall not make use of intrinsic procedures other than those described in
- 31 Clause 13.

32 12.6.2 Procedures defined by subprograms

1 12.6.2.1 General

2 A subprogram defines one or more procedures. A procedure is defined by the initial SUBROUTINE or
3 FUNCTION statement, and each ENTRY statement defines an additional procedure (12.6.2.6).

4 A subprogram is specified to be elemental (12.8), pure (12.7), recursive, or a separate module subprogram
5 (12.6.2.5) by a *prefix-spec* in its initial SUBROUTINE or FUNCTION statement.

6 R1225 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

7 R1226 *prefix-spec* **is** *declaration-type-spec*

8 **or** ELEMENTAL

9 **or** IMPURE

10 **or** MODULE

11 **or** PURE

12 **or** RECURSIVE

13 C1237 (R1225) A *prefix* shall contain at most one of each *prefix-spec*.

14 C1238 (R1225) A *prefix* shall not specify both PURE and IMPURE.

15 C1239 (R1225) A *prefix* shall not specify both ELEMENTAL and RECURSIVE.

16 C1240 (R1225) A *prefix* shall not specify ELEMENTAL if *proc-language-binding-spec* appears in the
17 *function-stmt* or *subroutine-stmt*.

18 C1241 (R1225) MODULE shall appear only within the *function-stmt* or *subroutine-stmt* of a module
19 subprogram or of an interface body that is declared in the scoping unit of a module or submodule.

20 C1242 (R1225) If MODULE appears within the *prefix* in a module subprogram, an accessible module
21 procedure interface having the same name as the subprogram shall be declared in the module
22 or submodule in which the subprogram is defined, or shall be declared in an ancestor of that
23 program unit.

24 C1243 (R1225) If MODULE appears within the *prefix* in a module subprogram, the subprogram shall
25 specify the same characteristics and dummy argument names as its corresponding (12.6.2.5)
26 module procedure interface body.

27 C1244 (R1225) If MODULE appears within the *prefix* in a module subprogram and a binding label
28 is specified, it shall be the same as the binding label specified in the corresponding module
29 procedure interface body.

30 C1245 (R1225) If MODULE appears within the *prefix* in a module subprogram, RECURSIVE shall
31 appear if and only if RECURSIVE appears in the *prefix* in the corresponding module procedure
32 interface body.

33 The RECURSIVE *prefix-spec* shall appear if any procedure defined by the subprogram directly or indi-
34 rectly invokes itself or any other procedure defined by the subprogram.

35 If the *prefix-spec* PURE appears, or the *prefix-spec* ELEMENTAL appears and IMPURE does not appear,
36 the subprogram is a pure subprogram and shall meet the additional constraints of 12.7.

37 If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet
38 the additional constraints of 12.8.1.

39 12.6.2.2 Function subprogram

40 A **function subprogram** is a subprogram that has a **FUNCTION statement** as its first statement.

- 1 R1227 *function-subprogram* **is** *function-stmt*
 2 [*specification-part*]
 3 [*execution-part*]
 4 [*internal-subprogram-part*]
 5 *end-function-stmt*
- 6 R1228 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
 7 ■ ([*dummy-arg-name-list*]) [*suffix*]
- 8 C1246 (R1228) If RESULT appears, *result-name* shall not be the same as *function-name* and shall not
 9 be the same as the *entry-name* in any ENTRY statement in the subprogram.
- 10 C1247 (R1228) If RESULT appears, the *function-name* shall not appear in any specification statement
 11 in the scoping unit of the function subprogram.
- 12 R1229 *proc-language-binding-spec* **is** *language-binding-spec*
- 13 C1248 (R1229) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the
 14 *function-stmt* or *subroutine-stmt* of an internal procedure, or of an interface body for an abstract
 15 interface or a dummy procedure.

J3 internal note

Unresolved Technical Issue 103

Should a redundant NAME=" be (unnecessarily) required?

Paper 06-352r3 takes what we in the U.K. call a “belt and braces” approach. It adds a constraint to force the user to specify NAME=" (no binding label), and just in case it adds a specification that if NAME= is missing entirely, there is no binding label.

That is not a good way of proceeding.

It is inconsistent with existing practice (dummy procedures).

With dummy procedures, the user not only has an “automatic lack of binding label”, he is not *allowed* to specify NAME=. This is certainly the most friendly way to go.

There are two possibilities:

1. automatic lack of binding name, as done for dummy procedures (and procedure pointers);
2. manual prohibition of binding name, forcing NAME=".

Item (1) is certainly friendlier, and consistent with what we’ve already done.

The only point I can see in favour of (2) is the documentary value of making the user say NAME=" when it is the only option. I think that is pretty marginal value, but others might disagree.

To resolve this issue, pick the approach (1) or (2), and do

1. automatic lack of binding name: nothing, just remove the UTI;
2. manual prohibition of binding name:
 - remove the 15.5.2 text inserted by 06-352r3,
 - reinstate C1240 from 06-352r3, and
 - undo my change to C1239.

- 16 C1249 (R1229) If *proc-language-binding-spec* is specified for a procedure, each of the procedure’s dummy
 17 arguments shall be a nonoptional interoperable variable (15.3.5, 15.3.6) or a nonoptional interoper-
 18 erable procedure (15.3.7). If *proc-language-binding-spec* is specified for a function, the function
 19 result shall be an interoperable scalar variable.
- 20 R1230 *dummy-arg-name* **is** *name*

- 1 C1250 (R1230) A *dummy-arg-name* shall be the name of a dummy argument.
- 2 R1231 *suffix* **is** *proc-language-binding-spec* [*RESULT* (*result-name*)]
3 **or** *RESULT* (*result-name*) [*proc-language-binding-spec*]
- 4 R1232 *end-function-stmt* **is** *END* [*FUNCTION* [*function-name*]]
- 5 C1251 (R1227) An internal function subprogram shall not contain an *ENTRY* statement.
- 6 C1252 (R1227) An internal function subprogram shall not contain an *internal-subprogram-part*.
- 7 C1253 (R1232) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-*
8 *name* specified in the *function-stmt*.
- 9 The name of the function is *function-name*.
- 10 The type and type parameters (if any) of the result of the function defined by a function subprogram
11 may be specified by a type specification in the *FUNCTION* statement or by the name of the result
12 variable appearing in a type declaration statement in the declaration part of the function subprogram.
13 They shall not be specified both ways. If they are not specified either way, they are determined by
14 the implicit typing rules in force within the function subprogram. If the function result is an array,
15 allocatable, or a pointer, this shall be specified by specifications of the name of the result variable within
16 the function body. The specifications of the function result attributes, the specification of dummy
17 argument attributes, and the information in the procedure heading collectively define the characteristics
18 of the function (12.3.1).
- 19 If *RESULT* appears, the name of the result variable of the function is *result-name* and all occurrences
20 of the function name in *execution-part* statements in the scoping unit refer to the function itself. If
21 *RESULT* does not appear, the result variable is *function-name* and all occurrences of the function name
22 in *execution-part* statements in the scoping unit are references to the result variable. The characteristics
23 (12.3.3) of the function result are those of the result variable. On completion of execution of the function,
24 the value returned is that of its result variable. If the function result is a pointer, the shape of the value
25 returned by the function is determined by the shape of the result variable when the execution of the
26 function is completed. If the result variable is not a pointer, its value shall be defined by the function.
27 If the function result is a pointer, on return the pointer association status of the result variable shall not
28 be undefined.

NOTE 12.45

The result variable is similar to any other variable 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 variable is used subsequently in the evaluation of the expression that invoked the function, an implementation may wish to defer releasing the storage occupied by that variable until after its value has been used in expression evaluation.

NOTE 12.46

An example of a recursive function is:

```

RECURSIVE FUNCTION CUMM_SUM (ARRAY) RESULT (C_SUM)
  REAL, INTENT (IN), DIMENSION (:) :: ARRAY
  REAL, DIMENSION (SIZE (ARRAY)) :: C_SUM
  INTEGER N
  N = SIZE (ARRAY)
  IF (N <= 1) THEN
    C_SUM = ARRAY
  ELSE

```

NOTE 12.46 (cont.)

```

      N = N / 2
      C_SUM (:N) = CUMM_SUM (ARRAY (:N))
      C_SUM (N+1:) = C_SUM (N) + CUMM_SUM (ARRAY (N+1:))
    END IF
  END FUNCTION CUMM_SUM

```

NOTE 12.47

The following is an example of the declaration of an interface body with the BIND attribute, and a reference to the procedure declared.

```

USE, INTRINSIC :: ISO_C_BINDING

INTERFACE
  FUNCTION JOE (I, J, R) BIND(C,NAME="FrEd")
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(C_INT) :: JOE
    INTEGER(C_INT), VALUE :: I, J
    REAL(C_FLOAT), VALUE :: R
  END FUNCTION JOE
END INTERFACE

INT = JOE(1_C_INT, 3_C_INT, 4.0_C_FLOAT)
END PROGRAM

```

The invocation of the function JOE results in a reference to a function with a binding label "FrEd". FrEd may be a C function described by the C prototype

```
int FrEd(int n, int m, float x);
```

1 12.6.2.3 Subroutine subprogram

2 A **subroutine subprogram** is a subprogram that has a **SUBROUTINE statement** as its first
3 statement.

4 R1233 *subroutine-subprogram* **is** *subroutine-stmt*
5 [*specification-part*]
6 [*execution-part*]
7 [*internal-subprogram-part*]
8 *end-subroutine-stmt*

9 R1234 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name* ■
10 ■ [([*dummy-arg-list*]) [*proc-language-binding-spec*]]

11 C1254 (R1234) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

12 R1235 *dummy-arg* **is** *dummy-arg-name*
13 **or** *

14 R1236 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

15 C1255 (R1233) An internal subroutine subprogram shall not contain an ENTRY statement.

- 1 C1256 (R1233) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.
 2 C1257 (R1236) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the
 3 *subroutine-name* specified in the *subroutine-stmt*.
 4 The name of the subroutine is *subroutine-name*.

5 12.6.2.4 Instances of a subprogram

- 6 When a procedure defined by a subprogram is invoked, an **instance** of that subprogram is created.
 7 Execution begins with the first executable construct following the FUNCTION, SUBROUTINE, or
 8 ENTRY statement specifying the name of the procedure invoked.
 9 When a statement function is invoked, an instance of that statement function is created.
 10 When execution of an instance completes it ceases to exist.
 11 Each instance has an independent sequence of execution and an independent set of dummy arguments
 12 and local unsaved data objects. If an internal procedure or statement function in the subprogram is invoked
 13 by name from an instance of the subprogram or from an internal subprogram or statement function that
 14 has access to the entities of that instance, the created instance of the internal subprogram or statement
 15 function also has access to the entities of that instance of the host subprogram. If an internal procedure
 16 is invoked via a dummy procedure or procedure pointer, the internal procedure has access to the entities
 17 of the host instance of that dummy procedure or procedure pointer.
 18 All other entities are shared by all instances of the subprogram.

NOTE 12.48

The value of a saved data object appearing in one instance may have been defined in a previous instance or by initialization in a DATA statement or type declaration statement.

19 12.6.2.5 Separate module procedures

- 20 A **separate module procedure** is a module procedure defined by a *separate-module-subprogram*,
 21 by a *function-subprogram* whose initial statement contains the keyword MODULE, or by a *subroutine-*
 22 *subprogram* whose initial statement contains the keyword MODULE. Its interface is declared by a module
 23 procedure interface body (12.4.3.2) in the *specification-part* of the module or submodule in which the
 24 procedure is defined, or in an ancestor module or submodule.

25 R1237 *separate-module-subprogram* **is** *mp-subprogram-stmt*
 26 [*specification-part*]
 27 [*execution-part*]
 28 [*internal-subprogram-part*]
 29 *end-mp-subprogram-stmt*

30 R1238 *mp-subprogram-stmt* **is** MODULE PROCEDURE *procedure-name*
 31

32 R1239 *end-mp-subprogram-stmt* **is** END [PROCEDURE [*procedure-name*]]

- 33 C1258 (R1237) The *procedure-name* shall be the same as the name of an accessible module procedure
 34 interface that is declared in the module or submodule in which the *separate-module-subprogram*
 35 is defined, or is declared in an ancestor of that program unit.

- 36 C1259 (R1239) If a *procedure-name* appears in the *end-mp-subprogram-stmt*, it shall be identical to the
 37 *procedure-name* in the MODULE PROCEDURE statement.

1 The order, number, types, kind type parameters, and names of the dummy arguments in an ENTRY
 2 statement may differ from the order, number, types, kind type parameters, and names of the dummy
 3 arguments in the FUNCTION or SUBROUTINE statement in the containing subprogram.

4 Because an ENTRY statement defines an additional function or an additional subroutine, it is referenced
 5 in the same manner as any other function or subroutine (12.5).

6 In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear
 7 in an executable statement preceding that ENTRY statement, unless it also appears in a FUNCTION,
 8 SUBROUTINE, or ENTRY statement that precedes the executable statement.

9 In a subprogram, a dummy argument specified in an ENTRY statement shall not appear in an executable
 10 statement preceding that ENTRY statement, unless it also appears in a FUNCTION, SUBROUTINE,
 11 or ENTRY statement that precedes the executable statement.

12 In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear in the expression
 13 of a statement function unless the name is also a dummy argument of the statement function, appears in a FUNCTION
 14 or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

15 If a dummy argument appears in an executable statement, the execution of the executable statement is
 16 permitted during the execution of a reference to the function or subroutine only if the dummy argument
 17 appears in the dummy argument list of the procedure name referenced.

18 If a dummy argument is used in a specification expression to specify an array bound or character length
 19 of an object, the appearance of the object in a statement that is executed during a procedure reference
 20 is permitted only if the dummy argument appears in the dummy argument list of the procedure name
 21 referenced and it is present (12.5.2.13).

22 A scoping unit containing a reference to a procedure defined by an ENTRY statement may have access to
 23 an interface body for the procedure. The procedure header for the interface body shall be a FUNCTION
 24 statement for an entry in a function subprogram and shall be a SUBROUTINE statement for an entry
 25 in a subroutine subprogram.

26 The keyword RECURSIVE is not used in an ENTRY statement. Instead, the presence or absence of
 27 RECURSIVE in the initial SUBROUTINE or FUNCTION statement controls whether the procedure
 28 defined by an ENTRY statement is permitted to reference itself or another procedure defined by the
 29 subprogram.

30 The keywords PURE and IMPURE are not used in an ENTRY statement. Instead, the procedure
 31 defined by an ENTRY statement is pure if and only if the subprogram is a pure subprogram.

32 The keyword ELEMENTAL is not used in an ENTRY statement. Instead, the procedure defined by
 33 an ENTRY statement is elemental if and only if ELEMENTAL is specified in the SUBROUTINE or
 34 FUNCTION statement.

35 12.6.2.7 RETURN statement

36 R1241 *return-stmt* is RETURN [*scalar-int-expr*]

37 C1265 (R1241) The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.

38 C1266 (R1241) The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

39 Execution of the **RETURN statement** completes execution of the instance of the subprogram in which
 40 it appears. If the expression appears and has a value *n* between 1 and the number of asterisks in the dummy argument
 41 list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate
 42 return specifier in the actual argument list of the referenced procedure. If the expression is omitted or has a value outside

1 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
3 executing a RETURN statement with no expression.

4 **12.6.2.8 CONTAINS statement**

5 R1242 *contains-stmt* **is** CONTAINS

6 The **CONTAINS statement** separates the body of a main program, module, submodule, or subpro-
7 gram from any internal or module subprograms it may contain, or it introduces the type-bound procedure
8 part of a derived-type definition (4.5.2). The CONTAINS statement is not executable.

9 **12.6.3 Definition and invocation of procedures by means other than Fortran**

10 A procedure may be defined by means other than Fortran. The interface of a procedure defined by
11 means other than Fortran may be specified by an interface body or procedure declaration statement. If
12 the interface of such a procedure does not have a *proc-language-binding-spec*, the means by which the
13 procedure is defined are processor dependent. A reference to such a procedure is made as though it were
14 defined by an external subprogram.

15 If the interface of a procedure has a *proc-language-binding-spec*, the procedure is interoperable (15.5).

16 Interoperation with C functions is described in 15.5.

NOTE 12.50

For explanatory information on definition of procedures by means other than Fortran, see subclause C.9.2.

17 **12.6.4 Statement function**

18 A statement function is a function defined by a single statement.

19 R1243 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*

20 C1267 (R1243) The *primaries* of the *scalar-expr* shall be constants (literal and named), references to variables, references
21 to functions and function dummy procedures, and intrinsic operations. If *scalar-expr* contains a reference to a
22 function or a function dummy procedure, the reference shall not require an explicit interface, the function shall
23 not require an explicit interface unless it is an intrinsic function, the function shall not be a transformational
24 intrinsic, and the result shall be scalar. If an argument to a function or a function dummy procedure is an array,
25 it shall be an array name. If a reference to a statement function appears in *scalar-expr*, its definition shall have
26 been provided earlier in the scoping unit and shall not be the name of the statement function being defined.

27 C1268 (R1243) Named constants in *scalar-expr* shall have been declared earlier in the scoping unit or made accessible
28 by use or host association. If array elements appear in *scalar-expr*, the array shall have been declared as an array
29 earlier in the scoping unit or made accessible by use or host association.

30 C1269 (R1243) If a *dummy-arg-name*, variable, function reference, or dummy function reference is typed by the implicit
31 typing rules, its appearance in any subsequent type declaration statement shall confirm this implied type and
32 the values of any implied type parameters.

33 C1270 (R1243) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.

34 C1271 (R1243) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.

35 C1272 (R1243) Each variable reference in *scalar-expr* may be either a reference to a dummy argument of the statement
function or a reference to a variable accessible in the same scoping unit as the statement function statement.

- 1 The definition of a statement function with the same name as an accessible entity from the host shall be preceded by the
2 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
4 type parameters as the entity of the same name in the scoping unit containing the statement function.
- 5 A statement function shall not be supplied as a procedure argument.
- 6 The value of a statement function reference is obtained by evaluating the expression using the values of the actual arguments
7 for the values of the corresponding dummy arguments and, if necessary, converting the result to the declared type and
8 type parameters of the function.
- 9 A function reference in the scalar expression shall not cause a dummy argument of the statement function to become
10 redefined or undefined.

11 12.7 Pure procedures

12 A pure procedure is

- 13 (1) a pure intrinsic procedure (13.1),
- 14 (2) defined by a pure subprogram,
- 15 (3) a dummy procedure that has been specified to be PURE, or
- 16 (4) a statement function that references only pure functions.

17 A pure subprogram is a subprogram that has the *prefix-spec* PURE or that has the *prefix-spec* ELE-
18 MENTAL and does not have the *prefix-spec* IMPURE. The following additional constraints apply to
19 pure subprograms.

- 20 C1273 The *specification-part* of a pure function subprogram shall specify that all its nonpointer dummy
21 data objects have INTENT(IN).
- 22 C1274 The *specification-part* of a pure subroutine subprogram shall specify the intents of all its non-
23 pointer dummy data objects.
- 24 C1275 A local variable of a pure subprogram, or of a BLOCK construct within a pure subprogram,
25 shall not have the SAVE attribute.

NOTE 12.51

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; therefore, such initialization is also disallowed.

- 26 C1276 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are
27 pure.
- 28 C1277 If a procedure that is neither an intrinsic procedure nor a statement function is used in a context
29 that requires it to be pure, then its interface shall be explicit in the scope of that use. The
30 interface shall specify that the procedure is pure.
- 31 C1278 All internal subprograms in a pure subprogram shall be pure.
- 32 C1279 In a pure subprogram any designator with a base object that is in common or accessed by
33 host or use association, is a dummy argument of a pure function, is a dummy argument with
34 INTENT (IN) of a pure subroutine, or an object that is storage associated with any such variable,
35 shall not be used
- 36 (1) in a variable definition context (16.6.7),

- 1 (2) as the *data-target* in a *pointer-assignment-stmt*,
- 2 (3) as the *expr* corresponding to a component with the POINTER attribute in a *structure-*
3 *constructor*,
- 4 (4) as the *expr* of an intrinsic assignment statement in which the variable is of a derived type
5 if the derived type has a pointer component at any level of component selection, or
- 6 (5) as an actual argument associated with a dummy argument with INTENT (OUT) or IN-
7 TENT (INOUT) or with the POINTER attribute.

NOTE 12.52

Item 3 requires that processors be able to determine if entities with the PRIVATE attribute or with private components have a pointer component.

- 8 C1280 Any procedure referenced in a pure subprogram, including one referenced via a defined operation,
9 defined assignment, user-defined derived-type input/output, or finalization, shall be pure.
- 10 C1281 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-*
11 *stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.
- 12 C1282 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-*
13 *number* or *.
- 14 C1283 A pure subprogram shall not contain a *stop-stmt*.
- 15 C1284 A co-indexed object shall not appear in a variable definition context in a pure subprogram.
- 16 C1285 A pure subprogram shall not contain an image control statement (8.5.1).

NOTE 12.53

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 a *FORALL assignment-stmt* or a *DO CONCURRENT* construct, where there is no explicit order of evaluation.

The constraints on pure subprograms may appear complicated, but it is not necessary for a programmer to be intimately familiar with them. From the programmer's point of view, these constraints can be summarized as follows: a pure subprogram shall not contain any operation that could conceivably result in an assignment or pointer assignment to a common variable, a variable accessed by use or host association, or an INTENT (IN) dummy argument; nor shall a pure subprogram contain any operation that could conceivably perform any external file input/output or STOP operation. Note the use of the word conceivably; it is not sufficient for a pure subprogram merely to be side-effect free in practice. For example, a function that contains an assignment to a global variable but in a block that is not executed in any invocation of the function is nevertheless not a pure function. The exclusion of functions of this nature is required if strict compile-time checking is to be used.

It is expected that most library procedures will conform to the constraints required of pure procedures, and so can be declared pure and referenced in *FORALL* statements and constructs, *DO CONCURRENT* constructs, and within user-defined pure procedures.

NOTE 12.54

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),

NOTE 12.54 (cont.)

INTENT (INOUT), and pointer dummy arguments are permitted.

12.8 Elemental procedures

12.8.1 Elemental procedure declaration and interface

An **elemental procedure** is an elemental intrinsic procedure or a procedure that is defined by an elemental subprogram.

An elemental 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.

C1286 All dummy arguments of an elemental procedure shall be scalar non-co-array dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.

C1287 The result variable of an elemental function shall be scalar and shall not have the POINTER or ALLOCATABLE attribute.

C1288 In the scoping unit of an elemental subprogram, an object designator with a dummy argument as the base object shall not appear in a *specification-expr* except as the designator in a type parameter inquiry (6.1.4) or as the argument to one of the intrinsic functions BIT_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

NOTE 12.55

The restriction on dummy arguments in specification expressions is imposed primarily to facilitate optimization. An example of usage that is not permitted is

```

ELEMENTAL REAL FUNCTION F (A, N)
  REAL, INTENT (IN) :: A
  INTEGER, INTENT (IN) :: N
  REAL :: WORK_ARRAY(N)    ! Invalid
  ...
END FUNCTION F

```

An example of usage that is permitted is

```

ELEMENTAL REAL FUNCTION F (A)
  REAL, INTENT (IN) :: A
  REAL (SELECTED_REAL_KIND (PRECISION (A)*2)) :: WORK
  ...
END FUNCTION F

```

12.8.2 Elemental function actual arguments and results

If a generic name or a specific name is used to reference an elemental function, the shape of the result is the same as the shape of the actual argument with the greatest rank. If there are no actual arguments or the actual arguments are all scalar, the result is scalar. For those elemental functions that have more than one argument, all actual arguments shall be conformable. In the array case, the values of the elements, if any, of the result are the same as would have been obtained if the scalar function had been applied separately, in array element order, to corresponding elements of each array actual argument.

NOTE 12.56

An example of an elemental reference to the intrinsic function MAX:

if X and Y are arrays of shape (M, N),

MAX (X, 0.0, Y)

is an array expression of shape (M, N) whose elements have values

MAX (X(I, J), 0.0, Y(I, J)), I = 1, 2, ..., M, J = 1, 2, ..., N

1 12.8.3 Elemental subroutine actual arguments

2 An elemental subroutine is one that has only scalar dummy arguments, but may have array actual
3 arguments. In a reference to an elemental subroutine, either all actual arguments shall be scalar, or
4 all actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy arguments shall
5 be arrays of the same shape and the remaining actual arguments shall be conformable with them. In
6 the case that the actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy
7 arguments are arrays, the values of the elements, if any, of the results are the same as would be obtained
8 if the subroutine had been applied separately, in array element order, to corresponding elements of each
9 array actual argument.

10 In a reference to the intrinsic subroutine MVBITS, the actual arguments corresponding to the TO and
11 FROM dummy arguments may be the same variable and may be associated scalar variables or associated
12 array variables all of whose corresponding elements are associated. Apart from this, the actual arguments
13 in a reference to an elemental subroutine must satisfy the restrictions of 12.5.2.14.

13 Intrinsic procedures and modules

13.1 Classes of intrinsic procedures

There are four classes of intrinsic procedures: inquiry functions, elemental functions, transformational functions, and subroutines. Some intrinsic subroutines are elemental. Some intrinsic subroutines are collective.

An **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 inquiry function states otherwise, these arguments are permitted to be unallocated allocatables or pointers that are not associated. An **elemental intrinsic function** is one that is specified for scalar arguments, but may be applied to array arguments as described in 12.8. All other intrinsic functions are **transformational functions**; they almost all have one or more array arguments or an array result. All standard intrinsic functions are pure.

The subroutine `MOVE_ALLOC` and the elemental subroutine `MVBITS` are pure. No other standard intrinsic subroutine is pure.

A **collective subroutine** is one of the intrinsic subroutines named in 13.5.15. These subroutines accept an optional argument `TEAM` that identifies a team. If the `TEAM` argument is not present, the team consists of all images. The specified team shall include the invoking image. If it is invoked by one image of a team, it shall be invoked by the same statement on all images of the team. There is an implicit team synchronization at the beginning and end of the execution of a collective subroutine.

Generic names of standard intrinsic procedures are listed in 13.5. In most cases, generic functions accept arguments of more than one type and the type of the result is the same as the type of the arguments. **Specific names** of standard intrinsic functions with corresponding generic names are listed in 13.6.

If an intrinsic procedure is used as an actual argument to a procedure, its specific name shall be used and it may be referenced in the called procedure only with scalar arguments. If an intrinsic procedure does not have a specific name, it shall not be used as an actual argument (12.5.2.10).

Elemental intrinsic procedures behave as described in 12.8.

13.2 Arguments to intrinsic procedures

13.2.1 General rules

All intrinsic procedures may be invoked with either positional arguments or argument keywords (12.5). The descriptions in 13.5 through 13.7 give the argument keyword names and positional sequence for standard intrinsic procedures.

Many of the intrinsic procedures have optional arguments. These arguments are identified by the notation “optional” in the argument descriptions. In addition, the names of the optional arguments are enclosed in square brackets in description headings and in lists of procedures. The valid forms of reference for procedures with optional arguments are described in 12.5.2.

NOTE 13.1

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 13.2

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.

1 The dummy arguments of the specific intrinsic procedures in 13.6 have INTENT(IN). The dummy
 2 arguments of the generic intrinsic procedures in 13.7 have INTENT(IN) if the intent is not stated
 3 explicitly.

4 The actual argument associated with an intrinsic function dummy argument named KIND shall be a
 5 scalar integer initialization expression and its value shall specify a representation method for the function
 6 result that exists on the processor.

7 Intrinsic subroutines that assign values to arguments of type character do so in accordance with the
 8 rules of intrinsic assignment (7.2.1.3).

9 **13.2.2 The shape of array arguments**

10 Unless otherwise specified, the inquiry intrinsic functions accept array arguments for which the shape
 11 need not be defined. The shape of array arguments to transformational and elemental intrinsic functions
 12 shall be defined.

13 **13.2.3 Mask arguments**

14 Some array intrinsic functions have an optional MASK argument of type logical that is used by the
 15 function to select the elements of one or more arguments to be operated on by the function. Any
 16 element not selected by the mask need not be defined at the time the function is invoked.

17 The MASK affects only the value of the function, and does not affect the evaluation, prior to invoking
 18 the function, of arguments that are array expressions.

19 **13.2.4 Arguments to collective subroutines**

20 Each argument to a collective subroutine shall have the same shape on all images of the team. A co-array
 21 argument shall have the same bounds on all images of the team. An INTENT(IN) argument of type
 22 IMAGE_TEAM shall have a value that was constructed by corresponding invocations of FORM_TEAM
 23 for the team.

24 **13.3 Bit model**

25 The bit manipulation procedures and inquiry functions are described in terms of a model for the repre-
 26 sentation and behaviour of bits on a processor.

27 For an integer, a bit is defined to be a binary digit *w* located at position *k* of a nonnegative integer scalar
 28 object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{z-1} w_k \times 2^k$$

1 and for which w_k may have the value 0 or 1. An example of a model number compatible with the
2 examples used in 13.4 would have $z = 32$, thereby defining a 32-bit integer.

3 Using the notation of the formula above, the value of an object of type bits and kind z is represented as
4 the ordered sequence of bits with w_k the bit at position k . The rightmost bit is w_0 and the leftmost bit
5 is w_{z-1} . Such a bits object can be interpreted as a nonnegative integer with the value j .

6 An inquiry function BIT_SIZE is available to determine the parameter z of the model.

7 Effectively, this model defines an integer object to consist of z bits in sequence numbered from right
8 to left from 0 to $z - 1$. This model is valid only in the context of the use of such an object as the
9 argument or result of one of the bit manipulation procedures or the reduction functions IANY, IALL, or
10 IPARITY. In all other contexts, the model defined for an integer in 13.4 applies. In particular, whereas
11 the models are identical for $r = 2$ and $w_{z-1} = 0$, they do not correspond for $r \neq 2$ or $w_{z-1} = 1$ and the
12 interpretation of bits in such objects is processor dependent.

13 13.4 Numeric models

14 The numeric manipulation and inquiry functions are described in terms of a model for the representation
15 and behavior of numbers on a processor. The model has parameters that are determined so as to make
16 the model best fit the machine on which the program is executed.

17 The model set for integer i is defined by

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k$$

18 where r is an integer exceeding one, q is a positive integer, each w_k is a nonnegative integer less than r ,
19 and s is +1 or -1.

20 The model set for real x is defined by

$$x = \begin{cases} 0 \text{ or} \\ s \times b^e \times \sum_{k=1}^p f_k \times b^{-k} \end{cases},$$

21 where b and p are integers exceeding one; each f_k is a nonnegative integer less than b , with f_1 nonzero; s
22 is +1 or -1; and e is an integer that lies between some integer maximum $e_{\max}$ and some integer minimum
23 $e_{\min}$ inclusively. For $x = 0$, its exponent e and digits f_k are defined to be zero. The integer parameters
24 r and q determine the set of model integers and the integer parameters b , p , $e_{\min}$, and $e_{\max}$ determine
25 the set of model floating-point numbers. The parameters of the integer and real models are available
26 for each representation method of the integer and real types. The parameters characterize the set of
27 available numbers in the definition of the model. The floating-point manipulation functions (13.5.10)
28 and numeric inquiry functions (13.5.6) provide values of some parameters and other values related to
29 the models.

NOTE 13.3

Examples of these functions in 13.7 use the models

$$i = s \times \sum_{k=0}^{30} w_k \times 2^k$$

and

$$x = 0 \text{ or } s \times 2^e \times \left(\frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \leq e \leq 127$$

1 13.5 Standard generic intrinsic procedures

- 2 For all of the standard intrinsic procedures, the arguments shown are the names that shall be used for
3 argument keywords if the keyword form is used for actual arguments.

NOTE 13.4

For example, a reference to CMPLX may be written in the form CMPLX (A, B, M) or in the form CMPLX (Y = B, KIND = M, X = A).

NOTE 13.5

Many of the argument keywords have names that are indicative of their usage. For example:

KIND	Describes the kind type parameter of the result
STRING, STRING_A	An arbitrary character string
BACK	Controls the direction of string scan (forward or backward)
MASK	A mask that may be applied to the arguments
DIM	A selected dimension of an array argument

4 13.5.1 Numeric functions

5	ABS (A)	Absolute value
6	AIMAG (Z)	Imaginary part of a complex number
7	AINIT (A [, KIND])	Truncation to whole number
8	ANINT (A [, KIND])	Nearest whole number
9	CEILING (A [, KIND])	Least integer greater than or equal to number
10	CMPLX (X [, Y, KIND])	Conversion to complex type
11	CONJG (Z)	Conjugate of a complex number
12	DBLE (A)	Conversion to double precision real type
13	DIM (X, Y)	Positive difference
14	DPROD (X, Y)	Double precision real product
15	FLOOR (A [, KIND])	Greatest integer less than or equal to number
16	INT (A [, KIND])	Conversion to integer type
17	MAX (A1, A2 [, A3,...])	Maximum value
18	MIN (A1, A2 [, A3,...])	Minimum value
19	MOD (A, P)	Remainder function
20	MODULO (A, P)	Modulo function

1	NINT (A [, KIND])	Nearest integer
2	REAL (A [, KIND])	Conversion to real type
3	SIGN (A, B)	Transfer of sign

4 13.5.2 Mathematical functions

5	ACOS (X)	Arccosine
6	ACOSH (X)	Hyperbolic arccosine
7	ASIN (X)	Arcsine
8	ASINH (X)	Hyperbolic arcsine
9	ATAN (X) or ATAN (Y, X)	Arctangent
10	ATAN2 (Y, X)	Arctangent
11	ATANH (X)	Hyperbolic arctangent
12	BESSEL_J0 (X)	Bessel function of the first kind of order zero
13	BESSEL_J1 (X)	Bessel function of the first kind of order one
14	BESSEL_JN (N,X)	Bessel function of the first kind of order N
15	BESSEL_Y0 (X)	Bessel function of the second kind of order zero
16	BESSEL_Y1 (X)	Bessel function of the second kind of order one
17	BESSEL_YN (N, X)	Bessel function of the second kind of order N
18	COS (X)	Cosine
19	COSH (X)	Hyperbolic cosine
20	ERF (X)	Error function
21	ERFC (X)	Complementary error function
22	ERFC_SCALED (X)	Exponentially-scaled complementary error function
23		
24	EXP (X)	Exponential
25	GAMMA (X)	Gamma function
26	HYPOT (X, Y)	Euclidean distance function
27	LOG (X)	Natural logarithm
28	LOG_GAMMA (X)	Logarithm of absolute value of gamma function
29	LOG10 (X)	Common logarithm (base 10)
30	SIN (X)	Sine
31	SINH (X)	Hyperbolic sine
32	SQRT (X)	Square root
33	TAN (X)	Tangent
34	TANH (X)	Hyperbolic tangent

35 13.5.3 Character functions

36	ACHAR (I [, KIND])	Character in given position in ASCII collating sequence
37		
38	ADJUSTL (STRING)	Adjust left
39	ADJUSTR (STRING)	Adjust right
40	CHAR (I [, KIND])	Character in given position in processor collating sequence
41		
42	IACHAR (C [, KIND])	Position of a character in ASCII collating sequence
43		
44	ICHAR (C [, KIND])	Position of a character in processor collating sequence
45		
46	INDEX (STRING, SUBSTRING [, BACK, KIND])	Starting position of a substring
47	LEN_TRIM (STRING [, KIND])	Length without trailing blank characters
48	LGE (STRING_A, STRING_B)	Lexically greater than or equal
49	LGT (STRING_A, STRING_B)	Lexically greater than

1	LLE (STRING_A, STRING_B)	Lexically less than or equal
2	LLT (STRING_A, STRING_B)	Lexically less than
3	MAX (A1, A2 [, A3,...])	Maximum value
4	MIN (A1, A2 [, A3,...])	Minimum value
5	REPEAT (STRING, NCOPIES)	Repeated concatenation
6	SCAN (STRING, SET [, BACK, KIND])	Scan a string for a character in a set
7	TRIM (STRING)	Remove trailing blank characters
8	VERIFY (STRING, SET [, BACK, KIND])	Verify the set of characters in a string

9 13.5.4 Kind functions

10	BITS_KIND (X)	Bits kind type parameter value, compatible with the argument
11		
12	KIND (X)	Kind type parameter value
13	SELECTED_BITS_KIND (N)	Bits kind type parameter value, given number of bits
14		
15	SELECTED_CHAR_KIND (NAME)	Character kind type parameter value, given character set name
16		
17	SELECTED_INT_KIND (R)	Integer kind type parameter value, given range
18	SELECTED_REAL_KIND ([P, R, RADIX])	Real kind type parameter value, given precision, range, and radix
19		

20 13.5.5 Miscellaneous type conversion functions

21	BITS (A [, KIND])	Convert to bits type
22	LOGICAL (L [, KIND])	Convert between objects of type logical with different kind type parameters
23		
24	TRANSFER (SOURCE, MOLD [, SIZE])	Treat first argument as if of type of second argument
25		

26 13.5.6 Numeric inquiry functions

27	DIGITS (X)	Number of significant digits of the model
28	EPSILON (X)	Number that is almost negligible compared to one
29		
30	HUGE (X)	Largest number of the model
31	MAXEXPONENT (X)	Maximum exponent of the model
32	MINEXPONENT (X)	Minimum exponent of the model
33	PRECISION (X)	Decimal precision
34	RADIX (X)	Base of the model
35	RANGE (X)	Decimal exponent range
36	TINY (X)	Smallest positive number of the model

37 13.5.7 Array inquiry functions

38	COLBOUND (CO_ARRAY [, DIM, KIND])	Lower co-bounds of a co-array
39	CO_UBOUND (CO_ARRAY [, DIM, KIND])	Upper co-bounds of a co-array
40	LBOUND (ARRAY [, DIM, KIND])	Lower dimension bounds of an array
41	SHAPE (SOURCE [, KIND])	Shape of an array or scalar
42	SIZE (ARRAY [, DIM, KIND])	Total number of elements in an array
43	UBOUND (ARRAY [, DIM, KIND])	Upper dimension bounds of an array

1 13.5.8 Other inquiry functions

2	ALLOCATED (ARRAY) or	Allocation status
3	ALLOCATED (SCALAR)	
4	ASSOCIATED (POINTER [, TARGET])	Association status inquiry or comparison
5	BIT_SIZE (I)	Number of bits of the model
6	EXTENDS_TYPE_OF (A, MOLD)	Same dynamic type or an extension
7	IS_CONTIGUOUS	Contiguity inquiry
8	LEN (STRING [, KIND])	Length of a character entity
9	NEW_LINE (A)	Newline character
10	PRESENT (A)	Argument presence
11	SAME_TYPE_AS (A, B)	Same dynamic type
12	STORAGE_SIZE (A [, KIND])	Size in bits of an array element

12 13.5.9 Bit manipulation procedures

13	BTEST (I, POS)	Bit testing
14	DSHIFTL (I, J, SHIFT)	Combined left shift
15	DSHIFTR (I, J, SHIFT)	Combined right shift
16	IAND (I, J)	Bitwise AND
17	IBCLR (I, POS)	Clear bit
18	IBITS (I, POS, LEN)	Bit extraction
19	IBSET (I, POS)	Set bit
20	IEOR (I, J)	Exclusive OR
21	IOR (I, J)	Inclusive OR
22	ISHFT (I, SHIFT)	Logical shift
23	ISHFTC (I, SHIFT [, SIZE])	Circular shift
24	LEADZ (I)	Number of leading zero bits
25	MASKL (I [, KIND])	Left justified bit mask
26	MASKR (I [, KIND])	Right justified bit mask
27	MERGE_BITS (I, J, MASK)	Merge bits under mask
28	MVBITS (FROM, FROMPOS, LEN, TO,	Copies bits from one integer to another
29	TOPOS)	
30	NOT (I)	Bitwise complement
31	POPCNT (I)	Number of one bits
32	POPPAR (I)	Parity of one bits
33	SHIFTA (I, SHIFT)	Right shift with fill
34	SHIFTL (I, SHIFT)	Left shift
35	SHIFTR (I, SHIFT)	Right shift
36	TRAILZ (I)	Number of trailing zero bits

36 13.5.10 Floating-point manipulation functions

37	EXPONENT (X)	Exponent part of a model number
38	FRACTION (X)	Fractional part of a number
39	NEAREST (X, S)	Nearest different processor number in given
40		direction
41	RRSPACING (X)	Reciprocal of the relative spacing of model
42		numbers near given number
43	SCALE (X, I)	Multiply a real by its base to an integer power
44	SET_EXPONENT (X, I)	Set exponent part of a number
45	SPACING (X)	Absolute spacing of model numbers near given
46		number

1 13.5.11 Vector and matrix multiply functions

2	DOT_PRODUCT (VECTOR_A,	Dot product of two rank-one arrays
3	VECTOR_B)	
3	MATMUL (MATRIX_A, MATRIX_B)	Matrix multiplication

4 13.5.12 Array reduction functions

5	ALL (MASK [, DIM])	True if all values are true
6	ANY (MASK [, DIM])	True if any value is true
7	COUNT (MASK [, DIM, KIND])	Number of true elements in an array
8	IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY [, MASK])	Bitwise AND of array elements
9	IANY (ARRAY, DIM [, MASK]) or IANY (ARRAY [, MASK])	Bitwise OR of array elements
10	IPARITY (ARRAY, DIM [, MASK]) or IPARITY (ARRAY [, MASK])	Bitwise exclusive OR of array elements
11	MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])	Maximum value in an array
12	MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])	Minimum value in an array
13	NORM2 (X)	L_2 norm of an array
14	PARITY (MASK [, DIM])	True if an odd number of values is true
15	PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])	Product of array elements
16	SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])	Sum of array elements

17 13.5.13 Array construction functions

18	CSHIFT (ARRAY, SHIFT [, DIM])	Circular shift
19	EOSHIFT (ARRAY, SHIFT [, BOUNDARY,	End-off shift
20	DIM])	
21	MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
22	PACK (ARRAY, MASK [, VECTOR])	Pack an array into an array of rank one under a mask
23	RESHAPE (SOURCE, SHAPE[, PAD,	Reshape an array
24	ORDER])	
25	SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
26	TRANSPOSE (MATRIX)	Transpose of an array of rank two
27	UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask

28 13.5.14 Array location functions

29	FINDLOC (ARRAY, VALUE, DIM, [,	Location of value in an array
30	MASK, KIND, BACK]) or FINDLOC (ARRAY, VALUE, [, MASK,	
31	KIND, BACK])	
31	MAXLOC (ARRAY, DIM [, MASK, KIND,	Location of a maximum value in an array
31	BACK]) or MAXLOC (ARRAY [,	
31	MASK, KIND, BACK])	
31	MINLOC (ARRAY, DIM [, MASK, KIND,	Location of a minimum value in an array
31	BACK]) or MINLOC (ARRAY [,	
31	MASK, KIND, BACK])	

1 13.5.15 Collective subroutines

2	CO_ALL (MASK, RESULT [, TEAM])	Whether all values are true
3	CO_ANY (MASK, RESULT [, TEAM])	Whether any value is true
4	CO_COUNT (MASK, RESULT [, TEAM])	Number of true elements in a co-array
5	CO_MAXLOC (CO_ARRAY, RESULT [, TEAM])	Image indices of maximum values
6	CO_MAXVAL (CO_ARRAY, RESULT [, TEAM])	Maximum values
7	CO_MINLOC (CO_ARRAY, RESULT [, TEAM])	Image indices of minimum values
8	CO_MINVAL (CO_ARRAY, RESULT [, TEAM])	Minimum values
9	CO_PRODUCT (CO_ARRAY, RESULT [, TEAM])	Products of elements
10	CO_SUM (CO_ARRAY, RESULT [, TEAM])	Sums of elements

11 13.5.16 Null function

12	NULL ([MOLD])	Returns disassociated or unallocated result

13 13.5.17 Allocation transfer procedure

14	MOVE_ALLOC (FROM, TO)	Moves an allocation from one allocatable object to another

16 13.5.18 Random number subroutines

17	RANDOM_NUMBER (HARVEST)	Returns pseudorandom number
18	RANDOM_SEED ([SIZE, PUT, GET])	Initializes or restarts the pseudorandom number generator

20 13.5.19 System environment procedures

21	COMMAND_ARGUMENT_COUNT ()	Number of command arguments
22	CPU_TIME (TIME)	Obtain processor time
23	DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])	Obtain date and time
24	EXECUTE_COMMAND_LINE (COMMAND, [WAIT, EXITSTAT, CMDSTAT, CMDMSG])	Execute a command line
25	FORM_TEAM (TEAM, IMAGES)	Forms a team of images
26	GET_COMMAND ([COMMAND, LENGTH, STATUS])	Returns entire command
27	GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])	Returns a command argument
28	GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])	Obtain the value of an environment variable
29	IMAGE_INDEX (CO_ARRAY, SUB)	Index of an image
30	IS_IOSTAT_END (I)	Test for end-of-file value
31	IS_IOSTAT_EOR (I)	Test for end-of-record value
32	NUM_IMAGES ()	Number of images
33	SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])	Obtain data from the system clock

1	TEAM_IMAGES (TEAM)	Indices of the images in a team
2	THIS_IMAGE ()	Index of the invoking image
3	THIS_IMAGE (CO_ARRAY [, DIM])	Co-subscript list

4 13.6 Specific names for standard intrinsic functions

5 Except for AMAX0, AMIN0, MAX1, and MIN1, the result type of the specific function is the same as the
6 result type of the corresponding generic function would be if it were invoked with the same arguments
7 as the specific function.

Specific Name	Generic Name	Argument Type
ABS	ABS	default real
ACOS	ACOS	default real
AIMAG	AIMAG	default complex
AINIT	AINIT	default real
ALOG	LOG	default real
ALOG10	LOG10	default real
• AMAX0 (...)	REAL (MAX (...))	default integer
• AMAX1	MAX	default real
• AMIN0 (...)	REAL (MIN (...))	default integer
• AMIN1	MIN	default real
AMOD	MOD	default real
ANINT	ANINT	default real
ASIN	ASIN	default real
ATAN (X)	ATAN	default real
ATAN2	ATAN2	default real
CABS	ABS	default complex
CCOS	COS	default complex
CEXP	EXP	default complex
• CHAR	CHAR	default integer
CLOG	LOG	default complex
CONJG	CONJG	default complex
COS	COS	default real
COSH	COSH	default real
CSIN	SIN	default complex
CSQRT	SQRT	default complex
DABS	ABS	double precision real
DACOS	ACOS	double precision real
DASIN	ASIN	double precision real
DATAN	ATAN	double precision real
DATAN2	ATAN2	double precision real
DCOS	COS	double precision real
DCOSH	COSH	double precision real
DDIM	DIM	double precision real
DEXP	EXP	double precision real
DIM	DIM	default real
DINT	AINIT	double precision real
DLOG	LOG	double precision real
DLOG10	LOG10	double precision real
• DMAX1	MAX	double precision real
• DMIN1	MIN	double precision real
DMOD	MOD	double precision real
DNINT	ANINT	double precision real

Specific Name	Generic Name	Argument Type
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
• FLOAT	REAL	default integer
IABS	ABS	default integer
• ICHAR	ICHAR	default character
IDIM	DIM	default integer
• IDINT	INT	double precision real
IDNINT	NINT	double precision real
• IFIX	INT	default real
INDEX	INDEX	default character
• INT	INT	default real
ISIGN	SIGN	default integer
LEN	LEN	default character
• 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
MOD	MOD	default integer
NINT	NINT	default real
• REAL	REAL	default integer
SIGN	SIGN	default real
SIN	SIN	default real
SINH	SINH	default real
• SNGL	REAL	double precision real
SQRT	SQRT	default real
TAN	TAN	default real
TANH	TANH	default real

- 1 A specific intrinsic function marked with a bullet (•) shall not be used as an actual argument or as a
2 target in a procedure pointer assignment statement.

3 13.7 Specifications of the standard intrinsic procedures

- 4 Detailed specifications of the standard generic intrinsic procedures are provided here in alphabetical
5 order.

- 6 The types and type parameters of standard intrinsic procedure arguments and function results are de-
7 termined by these specifications. The “Argument(s)” paragraphs specify requirements on the actual
8 arguments of the procedures. The result characteristics are sometimes specified in terms of the char-
9 acteristics of dummy arguments. A program is prohibited from invoking an intrinsic procedure under
10 circumstances where a value to be returned in a subroutine argument or function result is outside the

1 range of values representable by objects of the specified type and type parameters, unless the intrinsic
 2 module IEEE_ARITHMETIC (clause 14) is accessible and there is support for an infinite or a NaN
 3 result, as appropriate. If an infinite result is returned, the flag IEEE_OVERFLOW or IEEE_DIVIDE_-
 4 BY_ZERO shall signal; if a NaN result is returned, the flag IEEE_INVALID shall signal. If all results
 5 are normal, these flags shall have the same status as when the intrinsic procedure was invoked.

6 **13.7.1 ABS (A)**

7 **Description.** Absolute value.

8 **Class.** Elemental function.

9 **Argument.** A shall be of type integer, real, or complex.

10 **Result Characteristics.** The same as A except that if A is complex, the result is real.

11 **Result Value.** If A is of type integer or real, the value of the result is $|A|$; if A is complex with value
 12 (x, y) , the result is equal to a processor-dependent approximation to $\sqrt{x^2 + y^2}$ computed without undue
 13 overflow or underflow.

14 **Example.** ABS ((3.0, 4.0)) has the value 5.0 (approximately).

15 **13.7.2 ACHAR (I [, KIND])**

16 **Description.** Returns the character in a specified position of the ASCII collating sequence. It is the
 17 inverse of the IACHAR function.

18 **Class.** Elemental function.

19 **Arguments.**

20 I shall be of type integer or bits. If it is of type bits, it is interpreted as a nonnegative
 integer as described in 13.3.

21 KIND (optional) shall be a scalar integer initialization expression.

22 **Result Characteristics.** Character of length one. 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 character type.

24 **Result Value.** If I has a value in the range $0 \leq I \leq 127$, the result is the character in position I of
 25 the ASCII collating sequence, provided the processor is capable of representing that character in the
 26 character type of the result; otherwise, the result is processor dependent. ACHAR (IACHAR (C)) shall
 27 have the value C for any character C capable of representation in the default character type.

28 **Examples.** ACHAR (88) has the value 'X'. ACHAR (Z'41') has the value 'A'.

29 **13.7.3 ACOS (X)**

30 **Description.** Arccosine (inverse cosine) function.

31 **Class.** Elemental function.

32 **Argument.** X shall be of type real with a value that satisfies the inequality $|X| \leq 1$, or of type complex.

33 **Result Characteristics.** Same as X.

34 **Result Value.** The result has a value equal to a processor-dependent approximation to $\arccos(X)$. If it
 35 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 $0 \leq \text{REAL}(\text{ACOS}(X)) \leq \pi$.

1 **Example.** ACOS (0.54030231) has the value 1.0 (approximately).

2 **13.7.4 ACOSH (X)**

3 **Description.** Inverse hyperbolic cosine function.

4 **Class.** Elemental function.

5 **Argument.** X shall be of type real or complex.

6 **Result Characteristics.** Same as X.

7 **Result Value.** The result has a value equal to a processor-dependent approximation to the inverse
8 hyperbolic cosine function of X. If the result is complex the imaginary part is expressed in radians and
9 lies in the range $0 \leq \text{AIMAG}(\text{ACOSH}(X)) \leq \pi$

10 **Example.** ACOSH(1.5430806) has the value 1.0 (approximately).

11 **13.7.5 ADJUSTL (STRING)**

12 **Description.** Adjust to the left, removing leading blanks and inserting trailing blanks.

13 **Class.** Elemental function.

14 **Argument.** STRING shall be of type character.

15 **Result Characteristics.** Character of the same length and kind type parameter as STRING.

16 **Result Value.** The value of the result is the same as STRING except that any leading blanks have
17 been deleted and the same number of trailing blanks have been inserted.

18 **Example.** ADJUSTL (' WORD') has the value 'WORD '.

19 **13.7.6 ADJUSTR (STRING)**

20 **Description.** Adjust to the right, removing trailing blanks and inserting leading blanks.

21 **Class.** Elemental function.

22 **Argument.** STRING shall be of type character.

23 **Result Characteristics.** Character of the same length and kind type parameter as STRING.

24 **Result Value.** The value of the result is the same as STRING except that any trailing blanks have
25 been deleted and the same number of leading blanks have been inserted.

26 **Example.** ADJUSTR ('WORD ') has the value ' WORD'.

27 **13.7.7 AIMAG (Z)**

28 **Description.** Imaginary part of a complex number.

29 **Class.** Elemental function.

30 **Argument.** Z shall be of type complex.

31 **Result Characteristics.** Real with the same kind type parameter as Z.

32 **Result Value.** If Z has the value (x, y) , the result has the value y .

1 **Example.** AIMAG ((2.0, 3.0)) has the value 3.0.

2 13.7.8 AINT (A [, KIND])

3 **Description.** Truncation to a whole number.

4 **Class.** Elemental function.

5 **Arguments.**

6 A shall be of type real.

7 KIND (optional) shall be a scalar integer initialization expression.

8 **Result Characteristics.** The result is of type real. If KIND is present, the kind type parameter is
9 that specified by the value of KIND; otherwise, the kind type parameter is that of A.

10 **Result Value.** If $|A| < 1$, AINT (A) has the value 0; if $|A| \geq 1$, AINT (A) has a value equal to the
11 integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign
12 is the same as the sign of A.

13 **Examples.** AINT (2.783) has the value 2.0. AINT (-2.783) has the value -2.0.

14 13.7.9 ALL (MASK [, DIM])

15 **Description.** Determine whether all values are true in MASK along dimension DIM.

16 **Class.** Transformational function.

17 **Arguments.**

18 MASK shall be of type logical. It shall be an array.

DIM (optional) shall be scalar and of type integer with 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
19 dummy argument.

20 **Result Characteristics.** The result is of type logical with the same kind type parameter as MASK. It
21 is scalar if DIM is absent; otherwise, the result has rank $n - 1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$
22 $\dots, d_n)$ where $(d_1, d_2, \dots, d_n)$ is the shape of MASK.

23 **Result Value.**

24 *Case (i):* The result of ALL (MASK) has the value true if all elements of MASK are true or if
25 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
27 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
28 ALL (MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$).

29 **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 =
32 1) is [true, false, false] and ALL (B /= C, DIM = 2) is [false, false].

33 13.7.10 ALLOCATED (ARRAY) or ALLOCATED (SCALAR)

Description. Indicate whether an allocatable variable is allocated.

1

2 **Class.** Inquiry function.3 **Arguments.**

4 ARRAY shall be an allocatable array.

5 SCALAR shall be an allocatable scalar.

6 **Result Characteristics.** Default logical scalar.7 **Result Value.** The result has the value true if the argument (ARRAY or SCALAR) is allocated and
8 has the value false if the argument is unallocated.9 **13.7.11 ANINT (A [, KIND])**10 **Description.** Nearest whole number.11 **Class.** Elemental function.12 **Arguments.**

13 A shall be of type real.

14 KIND (optional) shall be a scalar integer initialization expression.

15 **Result Characteristics.** The result is of type real. If KIND is present, the kind type parameter is
16 that specified by the value of KIND; otherwise, the kind type parameter is that of A.17 **Result Value.** The result is the integer nearest A, or if there are two integers equally near A, the result
18 is whichever such integer has the greater magnitude.19 **Examples.** ANINT (2.783) has the value 3.0. ANINT (-2.783) has the value -3.0.20 **13.7.12 ANY (MASK [, DIM])**21 **Description.** Determine whether any value is true in MASK along dimension DIM.22 **Class.** Transformational function.23 **Arguments.**

24 MASK shall be of type logical. It shall be an array.

25 DIM (optional) shall be scalar and of type integer 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.26 **Result Characteristics.** The result is of type logical with the same kind type parameter as MASK. It
27 is scalar if DIM is absent; otherwise, the result has rank $n - 1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$
28 $\dots, d_n)$ where $(d_1, d_2, \dots, d_n)$ is the shape of MASK.29 **Result Value.**30 *Case (i):* The result of ANY (MASK) has the value true if any element of MASK is true and has
31 the value false if no elements are true or if MASK has size zero.32 *Case (ii):* If MASK has rank one, ANY(MASK,DIM) is equal to ANY(MASK). Otherwise, the
33 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
34 ANY(MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$).

1 Examples.

2 *Case (i):* The value of ANY ([.TRUE., .FALSE., .TRUE.]) is true.

3 *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 =
4 1) is [true, false, true] and ANY(B /= C, DIM = 2) is [true, true].

5 13.7.13 ASIN (X)

6 **Description.** Arcsine (inverse sine) function.

7 **Class.** Elemental function.

8 **Argument.** X shall be of type real with a value that satisfies the inequality $|X| \leq 1$, or of type complex.

9 **Result Characteristics.** Same as X.

10 **Result Value.** The result has a value equal to a processor-dependent approximation to $\arcsin(X)$. If it
11 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
12 real part is expressed in radians and lies in the range $-\pi/2 \leq \text{REAL}(\text{ASIN}(X)) \leq \pi/2$.

13 **Example.** ASIN (0.84147098) has the value 1.0 (approximately).

14 13.7.14 ASINH (X)

15 **Description.** Inverse hyperbolic sine function.

16 **Class.** Elemental function.

17 **Argument.** X shall be of type real or complex.

18 **Result Characteristics.** Same as X.

19 **Result Value.** The result has a value equal to a processor-dependent approximation to the inverse
20 hyperbolic sine function of X. If the result is complex the imaginary part is expressed in radians and lies
21 in the range $-\pi/2 \leq \text{AIMAG}(\text{ASINH}(X)) \leq \pi/2$.

22 **Example.** ASINH(1.1752012) has the value 1.0 (approximately).

23 13.7.15 ASSOCIATED (POINTER [, TARGET])

24 **Description.** Returns the association status of its pointer argument or indicates whether the pointer
25 is associated with the target.

26 **Class.** Inquiry function.

27 **Arguments.**

28 **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 shall be allowable as the *data-target* or *proc-target* in a pointer assignment statement
(optional) (7.2.2) in which **POINTER** is *data-pointer-object* or *proc-pointer-object*. If **TARGET**
29 is a pointer then its pointer association status shall not be undefined.

30 **Result Characteristics.** Default logical scalar.

31 **Result Value.**

- 1 *Case (i):* If TARGET is absent, the result is true if and only if POINTER is associated with a
 2 target.
- 3 *Case (ii):* If TARGET is present and is a procedure, the result is true if and only if POINTER is
 4 associated with TARGET.
- 5 *Case (iii):* If TARGET is present and is a procedure pointer, the result is true if and only if POINTER
 6 and TARGET are associated with the same procedure.
- 7 *Case (iv):* If TARGET is present and is a scalar target, the result is true if and only if TARGET is
 8 not a zero-sized storage sequence and POINTER is associated with a target that occupies
 9 the same storage units as TARGET.
- 10 *Case (v):* If TARGET is present and is an array target, the result is true if and only if POINTER is
 11 associated with a target that has the same shape as TARGET, is neither of size zero nor
 12 an array whose elements are zero-sized storage sequences, and occupies the same storage
 13 units as TARGET in array element order.
- 14 *Case (vi):* If TARGET is present and is a scalar pointer, the result is true if and only if POINTER
 15 and TARGET are associated, the targets are not zero-sized storage sequences, and they
 16 occupy the same storage units.
- 17 *Case (vii):* If TARGET is present and is an array pointer, the result is true if and only if POINTER
 18 and TARGET are both associated, have the same shape, are neither of size zero nor arrays
 19 whose elements are zero-sized storage sequences, and occupy the same storage units in
 20 array element order.

21 **Examples.** ASSOCIATED (CURRENT, HEAD) is true if CURRENT is associated with the target
 22 HEAD. After the execution of

23 A_PART => A (:N)

24 ASSOCIATED (A_PART, A) is true if N is equal to UBOUND (A, DIM = 1). After the execution of

25 NULLIFY (CUR); NULLIFY (TOP)

26 ASSOCIATED (CUR, TOP) is false.

27 **13.7.16 ATAN (X) or ATAN (Y, X)**

28 **Description.** Arctangent (inverse tangent) function.

29 **Class.** Elemental function.

30 **Arguments.**

31 Y shall be of type real.

32 X If Y is present, 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 is absent, X shall be of type
 real or complex.

33 **Result Characteristics.** Same as X.

34 **Result Value.** If Y is present, the result is the same as the result of ATAN2(Y,X). If Y is absent,
 35 the result has a value equal to a processor-dependent approximation to $\arctan(X)$ whose real part is
 36 expressed in radians and lies in the range $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$.

37 **Example.** ATAN (1.5574077) has the value 1.0 (approximately).

38 **13.7.17 ATAN2 (Y, X)**

1 **Description.** Arctangent (inverse tangent) function. The result is the principal value of the argument
2 of the nonzero complex number (X, Y).

3 **Class.** Elemental function.

4 **Arguments.**

5 Y shall be of type real.

X shall be of the same type and kind type parameter as Y. If Y has the value zero, X
6 shall not have the value zero.

7 **Result Characteristics.** Same as X.

8 **Result Value.** The result has a value equal to a processor-dependent approximation to the principal
9 value of the argument of the complex number (X, Y), expressed in radians. It lies in the range $-\pi \leq$
10 $\text{ATAN2}(Y, X) \leq \pi$ and is equal to a processor-dependent approximation to a value of $\arctan(Y/X)$ if
11 $X \neq 0$. If $Y > 0$, the result is positive. If $Y = 0$ and $X > 0$, the result is Y. If $Y = 0$ and $X < 0$, then the
12 result is approximately π if Y is positive real zero or the processor cannot distinguish between positive
13 and negative real zero, and approximately $-\pi$ if Y is negative real zero. If $Y < 0$, the result is negative.
14 If $X = 0$, the absolute value of the result is approximately $\pi/2$.

15 **Examples.** $\text{ATAN2}(1.5574077, 1.0)$ has the value 1.0 (approximately). If Y has the value $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$
16 and X has the value $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, the value of $\text{ATAN2}(Y, X)$ is approximately $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ -\frac{3\pi}{4} & -\frac{\pi}{4} \end{bmatrix}$.

17 13.7.18 ATANH (X)

18 **Description.** Inverse hyperbolic tangent function.

19 **Class.** Elemental function.

20 **Argument.** X shall be of type real or complex.

21 **Result Characteristics.** Same as X.

22 **Result Value.** The result has a value equal to a processor-dependent approximation to the inverse
23 hyperbolic tangent function of X. If the result is complex the imaginary part is expressed in radians and
24 lies in the range $-\pi/2 \leq \text{AIMAG}(\text{ATANH}(X)) \leq \pi/2$.

25 **Example.** $\text{ATANH}(0.76159416)$ has the value 1.0 (approximately).

26 13.7.19 BESSEL_J0 (X)

27 **Description.** Bessel function of the first kind of order zero.

28 **Class.** Elemental function.

29 **Argument.** X shall be of type real.

30 **Result Characteristics.** Same as X.

31 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel
32 function of the first kind of order zero of X.

33 **Example.** $\text{BESSEL\_J0}(1.0)$ has the value 0.765 (approximately).

13.7.20 BESSEL_J1 (X)

1 **Description.** Bessel function of the first kind of order one.

2 **Class.** Elemental function.

3 **Argument.** X shall be of type real.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel
6 function of the first kind of order one of X.

7 **Example.** BESSEL_J1 (1.0) has the value 0.440 (approximately).

8 **13.7.21 BESSEL_JN (N, X)**

9 **Description.** Bessel function of the first kind of order N.

10 **Class.** Elemental function.

11 **Arguments.**

12 N shall be of type integer and nonnegative.

13 X shall be of type real.

14 **Result Characteristics.** Same as X.

15 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel
16 function of the first kind of order N of X.

17 **Example.** BESSEL_JN (2, 1.0) has the value 0.115 (approximately).

18 **13.7.22 BESSEL_Y0 (X)**

19 **Description.** Bessel function of the second kind of order zero.

20 **Class.** Elemental function.

21 **Argument.** X shall be of type real. Its value shall be greater than zero.

22 **Result Characteristics.** Same as X.

23 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel
24 function of the second kind of order zero of X.

25 **Example.** BESSEL_Y0(1.0) has the value 0.088 (approximately).

26 **13.7.23 BESSEL_Y1 (X)**

27 **Description.** Bessel function of the second kind of order one.

28 **Class.** Elemental function.

29 **Argument.** X shall be of type real. Its value shall be greater than zero.

30 **Result Characteristics.** Same as X.

31 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel
32 function of the second kind of order one of X.

1 **Example.** BESSEL_Y1 (1.0) has the value -0.781 (approximately).

2 **13.7.24 BESSEL_YN (N, X)**

3 **Description.** Bessel function of the second kind of order N.

4 **Class.** Elemental function.

5 **Arguments.**

6 N shall be of type integer and nonnegative.

7 X shall be of type real. Its value shall be greater than zero.

8 **Result Characteristics.** Same as X.

9 **Result Value.** The result has a value equal to a processor-dependent approximation to the Bessel
10 function of the second kind of order N of X.

11 **Example.** BESSEL_YN (2, 1.0) has the value -1.651 (approximately).

12 **13.7.25 BIT_SIZE (I)**

13 **Description.** Returns the number of bits z defined by the model of 13.3.

14 **Class.** Inquiry function.

15 **Argument.** I shall be of type integer or bits. It may be a scalar or an array.

16 **Result Characteristics.** Scalar integer. If I is of type integer the kind type parameter is that of I,
17 otherwise it is that of default integer type.

18 **Result Value.** If I is of type integer, the result has the value of the number of bits z of the model integer
19 defined for bit manipulation contexts in 13.3. If I is of type bits, the result has the value KIND(I).

20 **Examples.** BIT_SIZE (1) has the value 32 if z of the model is 32. BIT_SIZE(Z'00FF') has the value
21 16.

22 **13.7.26 BITS (A [, KIND])**

23 **Description.** Convert to bits type.

24 **Class.** Elemental function.

25 **Arguments.**

26 A shall be of type integer, real, complex, logical, or bits.

27 KIND (optional) shall be a scalar integer initialization expression.

28 **Result Characteristics.** Bits. If KIND is present, the kind type parameter is that specified by the
29 value of KIND; otherwise, the kind type parameter is BITS_KIND(A).

30 **Result Value.** If A is of type bits and the kind type parameter of A is the same as that of the result,
31 the result value is the value of A.

32 If A is of type bits with a smaller kind type parameter value than that of the result, the rightmost
33 KIND(A) bits of the result value are the same as those of A and the remaining bits of the result are 0.

- 1 If A is of type bits with a larger kind type parameter value than that of the result, the result value
 2 consists of the rightmost $\text{KIND}(\text{result})$ bits of A.
- 3 If A is not of type bits and $\text{BITS\_KIND}(A)$ is greater than or equal to the kind type parameter of the
 4 result, the result value consists of the rightmost $\text{KIND}(\text{result})$ bits of the physical representation of A.
- 5 If A is not of type bits and $\text{BITS\_KIND}(A)$ is less than or equal to the kind type parameter of the result,
 6 the rightmost bits of the result are those of the physical representation of A and the remaining bits of
 7 the result are 0.
- 8 **Examples.** $\text{BITS}(\text{Z}'AB', 16)$ has the value $\text{Z}'00AB'$. $\text{BITS}(-1)$ has the value $\text{Z}'FFFFFFFF'$ for a
 9 processor whose default integer representation is 32-bit two's-complement. $\text{BITS}(.TRUE., 5)$ has the
 10 value $\text{B}'00001'$ for a processor that represents the logical value true by setting the low order bit of the
 11 internal value and clearing the other bits. $\text{BITS}(X)$ has the value $\text{Z}'7F800000'$ if the value of X is an
 12 IEEE single precision positive infinity.

13 13.7.27 BITS_KIND (X)

14 **Description.** Return bits kind compatible with the argument.

15 **Class.** Inquiry function.

16 **Arguments.**

17 X shall be of type bits, integer, real, complex, or logical.

18 **Result Characteristics.** Default integer scalar.

19 **Result Value.** If X is of type bits, the result has the value $\text{KIND}(X)$. If X is of type default integer,
 20 default real, or default logical, the result has the value $\text{NUMERIC\_STORAGE\_SIZE}$ (13.8.2.13). If X is
 21 of type double precision or default complex, the result has the value $2 \times \text{NUMERIC\_STORAGE\_SIZE}$.
 22 Otherwise, the result value is the number of bits of storage used by the processor to represent scalar
 23 objects of that type and kind.

24 **Example.** The value of $\text{BITS\_KIND}(0)$ is 32 if the size of a numeric storage unit is 32 bits.

25 13.7.28 BTEST (I, POS)

26 **Description.** Tests a bit of an integer or bits value.

27 **Class.** Elemental function.

28 **Arguments.**

29 I shall be of type integer or bits.

30 POS shall be of type integer. It shall be nonnegative and be less than $\text{BIT\_SIZE}(I)$.

31 **Result Characteristics.** Default logical.

32 **Result Value.** The result has the value true if bit POS of I has the value 1 and has the value false if
 33 bit POS of I has the value 0. The model for the interpretation of an integer value as a sequence of bits
 34 is in 13.3.

35 **Examples.** $\text{BTEST}(8, 3)$ has the value true. If A has the value $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, the value of $\text{BTEST}(A, 2)$
 36 is $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$ and the value of $\text{BTEST}(2, A)$ is $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$.

1 BTEST (B'01000', 3) has the value true.

2 13.7.29 CEILING (A [, KIND])

3 **Description.** Returns the least integer greater than or equal to its argument.

4 **Class.** Elemental function.

5 **Arguments.**

6 A shall be of type real.

7 KIND (optional) shall be a scalar integer initialization expression.

8 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
9 value of KIND; otherwise, the kind type parameter is that of default integer type.

10 **Result Value.** The result has a value equal to the least integer greater than or equal to A.

11 **Examples.** CEILING (3.7) has the value 4. CEILING (-3.7) has the value -3.

12 13.7.30 CHAR (I [, KIND])

13 **Description.** Returns the character in a given position of the processor collating sequence associated
14 with the specified kind type parameter. It is the inverse of the ICHAR function.

15 **Class.** Elemental function.

16 **Arguments.**

17 I shall be of type integer or bits. If I is of type bits, it is interpreted as a non-negative
integer as described in 13.3. The value of I shall be in the range $0 \leq I \leq n - 1$, where
 n is the number of characters in the collating sequence associated with the specified
kind type parameter.

18 KIND (optional) shall be a scalar integer initialization expression.

19 **Result Characteristics.** Character of length one. If KIND is present, the kind type parameter is that
20 specified by the value of KIND; otherwise, the kind type parameter is that of default character type.

21 **Result Value.** The result is the character in position I of the collating sequence associated with the
22 specified kind type parameter. ICHAR (CHAR (I, KIND (C))) shall have the value I for $0 \leq I \leq n - 1$
23 and CHAR (ICHAR (C), KIND (C)) shall have the value C for any character C capable of representation
24 in the processor.

25 **Examples.** CHAR (88) has the value 'X' on a processor using the ASCII collating sequence for the
26 default character type. CHAR (Z'41') has the value 'A' on a processor using the ASCII collating
27 sequence.

28 13.7.31 CMPLX (X [, Y, KIND])

29 **Description.** Convert to complex type.

30 **Class.** Elemental function.

31 **Arguments.**

32 X shall be of type integer, real, or complex, or bits.

Y (optional) shall be of type integer or real, or bits. If X is of type complex, Y shall not be present, nor shall Y be associated with an optional dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. The result is of type complex. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default real type.

Result Value. If Y is absent and X is not complex, it is as if Y were present with the value zero. If X is complex, it is as if X were real with the value REAL (X, KIND) and Y were present with the value AIMAG (X, KIND). CMPLX (X, Y, KIND) has the complex value whose real part is REAL (X, KIND) and whose imaginary part is REAL (Y, KIND).

Example. CMPLX (-3) has the value (-3.0, 0.0).

13.7.32 CO_ALL (MASK, RESULT [, TEAM])

Description. Determine whether all values are true on a team of images.

Class. Collective subroutine.

Arguments.

MASK shall be a co-array of type logical. It may be a scalar or an array. It is an INTENT(IN) argument.

RESULT shall be of type logical and have the same shape as MASK. It is an INTENT(OUT) argument. If it is scalar, it is assigned the value true if the value of MASK is true on all the images of the team, and the value false otherwise. If it is an array, each element is assigned the value true if all corresponding elements of MASK are true on all the images of the team, and the value false otherwise.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_ALL is performed. If TEAM is not present, the team consists of all images.

Example. If the number of images is two and MASK is the array [true, false, true] on one image and [true, true, true] on the other image, the value of RESULT after executing the statement CALL CO_ALL (MASK, RESULT) is [true, false, true].

13.7.33 CO_ANY (MASK, RESULT [, TEAM])

Description. Determine whether any value is true on a team of images.

Class. Collective subroutine.

Arguments.

MASK shall be a co-array of type logical. It may be a scalar or an array. It is an INTENT(IN) argument.

RESULT shall be of type logical and have the same shape as MASK. It is an INTENT(OUT) argument. If it is scalar it is assigned the value true if any value of MASK is true on any image of the team, and false otherwise. If it is an array, each element is assigned the value true if any of the corresponding elements of MASK are true on any image of the team, and false otherwise.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_ANY is performed. If TEAM is not present, the team consists of all images.

Example. If the number of images is two and MASK is the array [true, false, false] on one image and [true, true, false] on the other image, the value of RESULT after executing the statement CALL CO_ANY (MASK, RESULT) is [true, true, false].

13.7.34 CO_COUNT (MASK, RESULT [, TEAM])

Description. Count the numbers of true elements on a team of images.

Class. Collective subroutine.

Arguments.

MASK shall be a co-array of type logical. It may be a scalar or an array. It is an INTENT(IN) argument.

RESULT shall be of type integer and have the same shape as MASK. It is an INTENT(OUT) argument. If it is scalar, it is assigned a value equal to the number of images of the team for which MASK has the value true. If it is an array, each element is assigned a value equal to the number of corresponding elements of MASK on the images of the team that have the value true.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_COUNT is performed. If TEAM is not present, the team consists of all images.

Example. If the number of images is two and MASK is the array [true, false, false] on one image and [true, true, false] on the other image, the value of result after executing the statement CALL CO_COUNT (MASK, RESULT) is [2, 1, 0].

13.7.35 CO_LBOUND (CO_ARRAY [, DIM, KIND])

Description. Returns all the lower co-bounds or a specified lower co-bound of a co-array.

Class. Inquiry function.

Arguments.

CO_ARRAY shall be a co-array and may be of any type. It may be a scalar or an array. If it is allocatable it shall be allocated.

DIM (optional) shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the co-rank of CO_ARRAY. The corresponding actual argument shall not be an optional dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default integer type. The result is scalar if DIM is present; otherwise, the result is an array of rank one and size n , where n is the co-rank of CO_ARRAY.

Result Value.

Case (i): CO_LBOUND (CO_ARRAY, DIM) has a value equal to the lower co-bound for co-subscript DIM of CO_ARRAY.

1 *Case (ii):* CO_LBOUND (CO_ARRAY) has a value whose i^{th} element is equal to
 2 CO_LBOUND (CO_ARRAY, i), for $i = 1, 2, \dots, n$, where n is the co-rank of CO_ARRAY.

3 **Examples.** If A is allocated by the statement ALLOCATE (A [2:3, 7:*)] then CO_LBOUND (A) is
 4 [2, 7] and CO_LBOUND (A, DIM=2) is 7.

5 **13.7.36 CO_MAXLOC (CO_ARRAY, RESULT [, TEAM])**

6 **Description.** Image indices of the maximum values of the elements on a team of images.

7 **Class.** Collective subroutine.

8 **Arguments.**

CO_ARRAY shall be a co-array of type integer, real, bits, or character. It may be a scalar or an
 9 array. It is an INTENT(IN) argument.

RESULT shall be of type integer and have the same shape as CO_ARRAY. It is an IN-
 TENT(OUT) argument. If it is scalar, it is assigned a value equal to the image
 index of the maximum value of CO_ARRAY on the images of the team; if more than
 one image has the maximum value, the smallest such image index is assigned. If
 RESULT is an array, each element of RESULT is assigned a value equal to the image
 index of the maximum value of all the corresponding elements of CO_ARRAY on the
 images of the team; if more than one image has the maximum value, the smallest such
 image index is assigned.

10 If CO_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 argument is applied.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument
 11 that specifies the team for which CO_MAXLOC is performed. If TEAM is not present,
 the team consists of all images.

12 **Example.** If the number of images is two and CO_ARRAY is the array [1, 5, 6] on one image and
 13 [4, 1, 6] on the other image, the value of RESULT after executing the statement
 14 CALL CO_MAXLOC (CO_ARRAY, RESULT) is [2, 1, 1].

15 **13.7.37 CO_MAXVAL (CO_ARRAY, RESULT [, TEAM])**

16 **Description.** Maximum values of the elements on a team of images.

17 **Class.** Collective subroutine.

18 **Arguments.**

CO_ARRAY shall be a co-array of type integer, real, bits, or character. It may be a scalar or an
 19 array. It is an INTENT(IN) argument.

RESULT shall be of the same type and type parameters as CO_ARRAY, and shall have the
 same shape as CO_ARRAY. It is an INTENT(OUT) argument. If it is scalar, it is
 assigned a value equal to the maximum value of CO_ARRAY on all the images of the
 team. If it is an array, each element is assigned a value equal to the maximum value
 of all the corresponding elements of CO_ARRAY on all the images of the team.
 If CO_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 argument is applied.
 20

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_MAXVAL is performed. If TEAM is not present, the team consists of all images.

Example. If the number of images is two and CO_ARRAY is the array [1, 5, 3] on one image and [4, 1, 6] on the other image, the value of RESULT after executing the statement
CALL CO_MAXVAL (CO_ARRAY, RESULT) is [4, 5, 6].

13.7.38 CO_MINLOC (CO_ARRAY, RESULT [, TEAM])

Description. Image indices of the minimum values of the elements on a team of images.

Class. Collective subroutine.

Arguments.

CO_ARRAY shall be a co-array of type integer, real, bits, or character. It may be a scalar or an array. It is an INTENT(IN) argument.

RESULT shall be of type integer and have the same shape as CO_ARRAY. It is an INTENT(OUT) argument. If it is scalar, it is assigned a value equal to the image index of the minimum value of CO_ARRAY on all the images of the team; if more than one image has the minimum value, the smallest such image index is assigned. If it is an array, each element is assigned a value equal to the image index of the minimum value of all the corresponding elements of CO_ARRAY on the images of the team; if more than one image has the minimum value, the smallest such image index is assigned.

If CO_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 argument is applied.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_MINLOC is performed. If TEAM is not present, the team consists of all images.

Example. If the number of images is two and CO_ARRAY is the array [1, 5, 6] on one image and [4, 1, 6] on the other image, the value of RESULT after executing the statement
CALL CO_MINLOC (ARRAY, RESULT) is [1, 2, 1].

13.7.39 CO_MINVAL (CO_ARRAY, RESULT [, TEAM])

Description. Minimum values of the elements on a team of images.

Class. Collective subroutine.

Arguments.

CO_ARRAY shall be a co-array of type integer, real, bits, or character. It may be a scalar or an array. It is an INTENT(IN) argument.

RESULT shall be of the same type and type parameters as CO_ARRAY, and shall have the same shape as CO_ARRAY. It is an INTENT(OUT) argument. If it is scalar it is assigned a value equal to the minimum value of CO_ARRAY on all the images of the team. If it is an array, each element is assigned a value equal to the minimum value of all the corresponding elements of CO_ARRAY on all the images of the team.

If CO_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 argument is applied.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_MINVAL is performed. If TEAM is not present, the team consists of all images.

Example. If the number of images is two and CO_ARRAY is the array [1, 5, 3] on one image and [4, 1, 6] on the other image, the value of RESULT after executing the statement CALL CO_MINVAL (CO_ARRAY, RESULT) is [1, 1, 3].

13.7.40 CO_PRODUCT (CO_ARRAY, RESULT [, TEAM])

Description. Products of elements on a team of images.

Class. Collective subroutine.

Arguments.

CO_ARRAY shall be a co-array of type integer, real, or complex. It may be a scalar or an array. It is an INTENT(IN) argument.

RESULT shall be of the same type and type parameters as CO_ARRAY, and shall have the same shape as CO_ARRAY. It is an INTENT(OUT) argument. If it is scalar, it is assigned a value equal to a processor-dependent and image-dependent approximation to the product of the values of CO_ARRAY on all the images of the team. If it is an array, each element is assigned a value equal to a processor-dependent and image-dependent approximation to the product of all the corresponding elements of CO_ARRAY on the images of the team.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_PRODUCT is performed. If TEAM is not present, the team consists of all images.

Example. If the number of images is two and CO_ARRAY is the array [1, 5, 3] on one image and [4, 1, 6] on the other image, the value of RESULT after executing the statement CALL CO_PRODUCT (CO_ARRAY, RESULT) is [4, 5, 18].

13.7.41 CO_SUM (CO_ARRAY, RESULT [, TEAM])

Description. Sums of elements on a team of images.

Class. Collective subroutine.

Arguments.

CO_ARRAY shall be a co-array of type integer, real, or complex. It may be a scalar or an array. It is an INTENT(IN) argument.

RESULT shall be of the same type and type parameters as CO_ARRAY, and shall have the same shape as CO_ARRAY. It is an INTENT(OUT) argument. If it is scalar, it is assigned a value equal to a processor-dependent and image-dependent approximation to the sum of the values of CO_ARRAY on all the images of the team. If it is an array, each element is assigned a value equal to a processor-dependent and image-dependent approximation to the sum of all the corresponding elements of CO_ARRAY on the images of the team.

TEAM (optional) shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(IN) argument that specifies the team for which CO_SUM is performed. If TEAM is not present, the team consists of all images.

1 **Example.** If the number of images is two and CO_ARRAY is the array [1, 5, 3] on one image and
 2 [4, 1, 6] on the other image, the value of RESULT after executing the statement
 3 CALL CO_SUM (CO_ARRAY, RESULT) is [5, 6, 9].

4 **13.7.42 CO_UBOUND (CO_ARRAY [, DIM, KIND])**

5 **Description.** Returns all the upper co-bounds or a specified upper co-bound of a co-array of co-rank
 6 greater than one.

7 **Class.** Inquiry function.

8 **Arguments.**

CO_ARRAY shall be a co-array of any type and of co-rank greater than one. It may be a scalar or
 9 an array. If it is allocatable it shall be allocated.

DIM (optional) d shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n - 1$, where
 10 n is the co-rank of CO_ARRAY. The corresponding actual argument shall not be an
 optional dummy argument.

J3 internal note

Unresolved Technical Issue 105

CO_UBOUND is inconvenient and inconsistent.

This is a very peculiar intrinsic. The processor knows jolly well how many array elements there are; in this it is not, in fact, analogous to the assumed-size array case. (Yes, some processors do know how big assumed-size arrays are, but not all and its not required by the standard – again, not analogous.)

The only way in which this is analogous to an assumed-size array is

1. The last co-dimension might, I repeat *might*, be a few images short of a whole column.
2. The use of an asterisk in the syntax. This is an irrelevance – asterisk is also used for assumed length type parameters (and one *is* permitted to ask about those).

The result shape of CO_UBOUND is also inconsistent with UBOUND. In fact the vector form of UBOUND is simply not permitted with an assumed-size array!

There is no doubt what the value of the co-bound of the last co-dimension is; for example, for the dummy argument REAL, ALLOCATABLE :: X(:, :, :, *) it is

```
CO_LBOUND(X,4) +
  CEILING (REAL (NUM_IMAGES()) /
    ((CO_UBOUND(X,1)-CO_LBOUND(X,1)+1)*
      CO_UBOUND(X,2)-CO_LBOUND(X,2)+1)*
      CO_UBOUND(X,3)-CO_LBOUND(X,3)+1))) - 1
```

Why are we even thinking of making the user write out this abomination?

J3 internal note (cont.)

Just tell him the answer!

P.S. Yes, in my mind there is no doubt that the co-bound for the last co-dimension is the ceiling and not the floor, because we say the technical term co-bound(s) means the bounds of the co-dimensions – that’s what bounds are (limits one cannot go beyond). The fact that a few images might be missing off the end doesn’t affect that.

P.P.S. The fact that the number of images, and thus the final upper co-bounds of all co-arrays, is determined by magic instead of by a Fortran statement that declares it, is neither here nor there; there’s no expression in an assumed-shape array declaration either.

P.P.P.S. Here we have the situation where providing the right answer would be a significant convenience to the user, and would make CO_UBOUND consistent with UBOUND to boot.

P.P.P.P.S. Paper 06-379 attempted to fix this, but there was a claim that this conflicted with [93:15], which says

The co-rank is equal to one plus the number of *upper-co-bounds*.

Well, it does not conflict. The technical term **co-bound** is defined in chapter two; the syntax term *upper-co-bound* is (a) spelled differently, and (b) even if the syntax term were simply *co-bound*, would not override that definition. (Allocatable coarrays don’t even have the syntax thingy, so it cannot be the definition of co-bound anyway. I’ll note that the original definition of co-bounds (in 06-007) was completely faulty (like, didn’t work at all) and that has probably influenced this confusion. (Of course paper 06-379 didn’t do the full and proper fix here, but that’s another story.)

1 KIND (optional) shall be a scalar integer initialization expression.

2 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
3 value of KIND; otherwise, the kind type parameter is that of default integer type. The result is scalar
4 if DIM is present; otherwise, the result is an array of rank one and size $n - 1$, where n is the co-rank of
5 CO_ARRAY.

6 **Result Value.**

7 *Case (i):* CO_UBOUND (CO_ARRAY, DIM) has a value equal to the upper co-bound for co-
8 subscript DIM of CO_ARRAY.

9 *Case (ii):* CO_UBOUND (CO_ARRAY) has a value whose i^{th} element is equal to
10 CO_UBOUND (CO_ARRAY, i), for $i = 1, 2, \dots, n - 1$, where n is the co-rank of CO-
11 ARRAY.

12 **Examples.** If A is allocated by the statement ALLOCATE (A [2:3, 0:7, *]) then
13 CO_UBOUND (A) is [3, 7] and CO_UBOUND (A, DIM=2) is 7.

14 **13.7.43 COMMAND_ARGUMENT_COUNT ()**

15 **Description.** Returns the number of command arguments.

16 **Class.** Inquiry function.

17 **Argument.** None.

18 **Result Characteristics.** Scalar default integer.

19 **Result Value.** The result value is equal to the number of command arguments available. If there are
20 no command arguments available or if the processor does not support command arguments, then the
21 result has the value zero. If the processor has a concept of a command name, the command name does
22 not count as one of the command arguments.

23 **Example.** See 13.7.73.

1 13.7.44 CONJG (Z)

2 **Description.** Conjugate of a complex number.

3 **Class.** Elemental function.

4 **Argument.** Z shall be of type complex.

5 **Result Characteristics.** Same as Z.

6 **Result Value.** If Z has the value (x, y) , the result has the value $(x, -y)$.

7 **Example.** CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

8 13.7.45 COS (X)

9 **Description.** Cosine function.

10 **Class.** Elemental function.

11 **Argument.** X shall be of type real or complex.

12 **Result Characteristics.** Same as X.

13 **Result Value.** The result has a value equal to a processor-dependent approximation to $\cos(X)$. If X is
14 of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a
15 value in radians.

16 **Example.** COS (1.0) has the value 0.54030231 (approximately).

17 13.7.46 COSH (X)

18 **Description.** Hyperbolic cosine function.

19 **Class.** Elemental function.

20 **Argument.** X shall be of type real or complex.

21 **Result Characteristics.** Same as X.

22 **Result Value.** The result has a value equal to a processor-dependent approximation to $\cosh(X)$. If X
23 is of type complex its imaginary part is regarded as a value in radians.

24 **Example.** COSH (1.0) has the value 1.5430806 (approximately).

25 13.7.47 COUNT (MASK [, DIM, KIND])

26 **Description.** Count the number of true elements of MASK along dimension DIM.

27 **Class.** Transformational function.

28 **Arguments.**

29 MASK shall be of type logical. It shall be an array.

DIM (optional) shall be scalar and of type integer 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
30 dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

1

2 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
 3 value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar
 4 if DIM is absent; otherwise, the result has rank $n - 1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$
 5 where $(d_1, d_2, \dots, d_n)$ is the shape of MASK.

6 **Result Value.**

7 *Case (i):* The result of COUNT (MASK) has a value equal to the number of true elements of MASK
 8 or has the value zero if MASK has size zero.

9 *Case (ii):* If MASK has rank one, COUNT (MASK, DIM) has a value equal to that of COUNT
 10 (MASK). Otherwise, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of
 11 COUNT (MASK, DIM) is equal to COUNT (MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1},$
 12 $\dots, s_n)$).

13 **Examples.**

14 *Case (i):* The value of COUNT ([.TRUE., .FALSE., .TRUE.]) is 2.

15 *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 =
 16 1) is [2, 0, 1] and COUNT (B /= C, DIM = 2) is [1, 2].

17 13.7.48 CPU_TIME (TIME)

18 **Description.** Returns the processor time.

19 **Class.** Subroutine.

20 **Argument.** TIME shall be scalar and of type real. It is an INTENT(OUT) argument that is assigned
 21 a processor-dependent approximation to the processor time in seconds. If the processor cannot return a
 22 meaningful time, a processor-dependent negative value is returned.

23 **Example.**

```
24      REAL T1, T2
25      ...
26      CALL CPU_TIME(T1)
27      ... ! Code to be timed.
28      CALL CPU_TIME(T2)
29      WRITE (*,*) 'Time taken by code was ', T2-T1, ' seconds'
```

30 writes the processor time taken by a piece of code.

NOTE 13.6

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.

NOTE 13.6 (cont.)

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.

1 13.7.49 CSHIFT (ARRAY, SHIFT [, DIM])

2 **Description.** Perform a circular shift on an array expression of rank one or perform circular shifts on
 3 all the complete rank one sections along a given dimension of an array expression of rank two or greater.
 4 Elements shifted out at one end of a section are shifted in at the other end. Different sections may be
 5 shifted by different amounts and in different directions.

6 **Class.** Transformational function.

7 **Arguments.**

8 **ARRAY** may be of any type. It shall be an array.

SHIFT shall be of type integer and shall be scalar if **ARRAY** has rank one; otherwise, it shall
 be scalar or of rank $n - 1$ and of shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$ where
 9 $(d_1, d_2, \dots, d_n)$ is the shape of **ARRAY**.

DIM (optional) shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is
 10 the rank of **ARRAY**. If **DIM** is omitted, it is as if it were present with the value 1.

11 **Result Characteristics.** The result is of the type and type parameters of **ARRAY**, and has the shape
 12 of **ARRAY**.

13 **Result Value.**

14 *Case (i):* If **ARRAY** has rank one, element i of the result is **ARRAY** $(1 + \text{MODULO}(i + \text{SHIFT}$
 15 $- 1, \text{SIZE}(\text{ARRAY}))$).

16 *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
 17 the result has a value equal to **CSHIFT** (**ARRAY** $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots,$
 18 $s_n)$, $sh, 1)$, where sh is **SHIFT** or **SHIFT** $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$.

19 **Examples.**

20 *Case (i):* If **V** is the array $[1, 2, 3, 4, 5, 6]$, the effect of shifting **V** circularly to the left by two
 21 positions is achieved by **CSHIFT** (**V**, **SHIFT** = 2) which has the value $[3, 4, 5, 6, 1, 2]$;
 22 **CSHIFT** (**V**, **SHIFT** = -2) achieves a circular shift to the right by two positions and has
 23 the value $[5, 6, 1, 2, 3, 4]$.

24 *Case (ii):* The rows of an array of rank two may all be shifted by the same amount or by different

25 amounts. If **M** is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, the value of

26 **CSHIFT** (**M**, **SHIFT** = -1, **DIM** = 2) is $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$, and the value of

27 **CSHIFT** (**M**, **SHIFT** = $[-1, 1, 0]$, **DIM** = 2) is $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$.

28 13.7.50 DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])

1 **Description.** Returns data about the real-time clock and date in a form compatible with the represen-
 2 tations defined in ISO 8601:1988.

3 **Class.** Subroutine.

4 **Arguments.**

DATE (optional) shall be scalar and of type default character. It is an INTENT (OUT) argument. It
 is assigned a value of the form *CCYYMMDD*, where *CC* is the century, *YY* is the
 year within the century, *MM* is the month within the year, and *DD* is the day within
 5 the month. If there is no date available, DATE is assigned all blanks.

TIME (optional) shall be scalar and of type default character. It is an INTENT (OUT) argument. It
 is assigned a value of the form *hhmmss.sss*, where *hh* is the hour of the day, *mm* is
 the minutes of the hour, and *ss.sss* is the seconds and milliseconds of the minute. If
 6 there is no clock available, TIME is assigned all blanks.

ZONE (optional) shall be scalar and of type default character. 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,
 7 respectively. If this information is not available, ZONE is assigned all blanks.

VALUES shall be of type default integer and of rank one. It is an INTENT (OUT) argument.
 8 (optional) Its size shall be at least 8. The values returned in VALUES are as follows:

VALUES (1) the year, including the century (for example, 1990), or -HUGE (0) if there is no date
 9 available;

10 VALUES (2) the month of the year, or -HUGE (0) if there is no date available;

11 VALUES (3) the day of the month, or -HUGE (0) if there is no date available;

VALUES (4) the time difference with respect to Coordinated Universal Time (UTC) in minutes,
 12 or -HUGE (0) if this information is not available;

13 VALUES (5) the hour of the day, in the range of 0 to 23, or -HUGE (0) if there is no clock;

14 VALUES (6) the minutes of the hour, in the range 0 to 59, or -HUGE (0) if there is no clock;

15 VALUES (7) the seconds of the minute, in the range 0 to 60, or -HUGE (0) if there is no clock;

VALUES (8) the milliseconds of the second, in the range 0 to 999, or -HUGE (0) if there is no
 16 clock.

17 **Example.**

18 INTEGER DATE_TIME (8)

19 CHARACTER (LEN = 10) BIG_BEN (3)

20 CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2), &
 21 BIG_BEN (3), DATE_TIME)

22 If run in Geneva, Switzerland on April 12, 1985 at 15:27:35.5 with a system configured for the local
 23 time zone, this sample would have assigned the value 19850412 to BIG_BEN (1), the value 152735.500
 24 to BIG_BEN (2), the value +0100 to BIG_BEN (3), and the value [1985, 4, 12, 60, 15, 27, 35, 500] to
 25 DATE_TIME.

NOTE 13.7

UTC is defined by ISO 8601:1988.

1 **13.7.51 DBLE (A)**

2 **Description.** Convert to double precision real type.

3 **Class.** Elemental function.

4 **Argument.** A shall be of type integer, real, complex, or bits.

5 **Result Characteristics.** Double precision real.

6 **Result Value.** The result has the value REAL (A, KIND (0.0D0)).

7 **Example.** DBLE (−3) has the value −3.0D0.

8 **13.7.52 DIGITS (X)**

9 **Description.** Returns the number of significant digits of the model representing numbers of the same
10 type and kind type parameter as the argument.

11 **Class.** Inquiry function.

12 **Argument.** X shall be of type integer or real. It may be a scalar or an array.

13 **Result Characteristics.** Default integer scalar.

14 **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
15 p are as defined in 13.4 for the model representing numbers of the same type and kind type parameter
16 as X.

17 **Example.** DIGITS (X) has the value 24 for real X whose model is as in Note 13.3.

18 **13.7.53 DIM (X, Y)**

19 **Description.** The difference $X-Y$ if it is positive; otherwise zero.

20 **Class.** Elemental function.

21 **Arguments.**

22 X shall be of type integer or real.

23 Y shall be of the same type and kind type parameter as X.

24 **Result Characteristics.** Same as X.

25 **Result Value.** The value of the result is $X-Y$ if $X>Y$ and zero otherwise.

26 **Example.** DIM (−3.0, 2.0) has the value 0.0.

27 **13.7.54 DOT_PRODUCT (VECTOR_A, VECTOR_B)**

28 **Description.** Performs dot-product multiplication of numeric or logical vectors.

29 **Class.** Transformational function.

1 **Arguments.**

2 VECTOR_A shall be of numeric type (integer, real, or complex) or of logical type. It shall be a rank-one array.

3 VECTOR_B shall be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It shall be a rank-one array. It shall be of the same size as VECTOR_A.

4 **Result Characteristics.** If the arguments are of numeric type, the type and kind type parameter of the result are those of the expression VECTOR_A * VECTOR_B determined by the types of the arguments according to 7.1.9.2. If the arguments are of type logical, the result is of type logical with the kind type parameter of the expression VECTOR_A .AND. VECTOR_B according to 7.1.9.2. The result is scalar.

9 **Result Value.**

10 *Case (i):* If VECTOR_A is of type integer or real, the result has the value SUM (VECTOR_A*VECTOR_B). If the vectors have size zero, the result has the value zero.

12 *Case (ii):* If VECTOR_A is of type complex, the result has the value SUM (CONJG (VECTOR_A)*VECTOR_B). If the vectors have size zero, the result has the value zero.

14 *Case (iii):* If VECTOR_A is of type logical, the result has the value ANY (VECTOR_A .AND. VECTOR_B). If the vectors have size zero, the result has the value false.

16 **Example.** DOT_PRODUCT ([1, 2, 3], [2, 3, 4]) has the value 20.

17 **13.7.55 DPROD (X, Y)**

18 **Description.** Double precision real product.

19 **Class.** Elemental function.

20 **Arguments.**

21 X shall be of type default real.

22 Y shall be of type default real.

23 **Result Characteristics.** Double precision real.

24 **Result Value.** The result has a value equal to a processor-dependent approximation to the product of X and Y.

26 **Example.** DPROD (-3.0, 2.0) has the value -6.0D0.

27 **13.7.56 DSHIFTL (I, J, SHIFT)**

28 **Description.** Performs a combined left shift.

29 **Class.** Elemental function.

30 **Arguments.**

31 I shall be of type integer or bits.

32 J shall be of the same type and kind as I.

33 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

34 **Result Characteristics.** Same as I.

1 **Result Value.** The rightmost SHIFT bits of the result value are the same as the leftmost bits of J,
 2 and the remaining bits of the result value are the same as the rightmost bits of I. This is equal to
 3 IOR (SHIFTL (I, SHIFT), SHIFTR (J, BIT_SIZE (J)–SHIFT)).

4 **Examples.** DSHIFTL (Z'01234567', Z'89ABCDEF', 8) has the value Z'23456789'.
 5 DSHIFTL (I, I, SHIFT) has the same result value as ISHFTC (I, SHIFT).

6 **13.7.57 DSHIFTR (I, J, SHIFT)**

7 **Description.** Performs a combined right shift.

8 **Class.** Elemental function.

9 **Arguments.**

10 I shall be of type integer or bits.
 11 J shall be of the same type and kind as I.
 12 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

13 **Result Characteristics.** Same as I.

14 **Result Value.** The leftmost SHIFT bits of the result value are the same as the rightmost bits of
 15 I, and the remaining bits of the result value are the same as the leftmost bits of J. This is equal to
 16 IOR (SHIFTL (I, BIT_SIZE (I)–SHIFT), SHIFTR (J, SHIFT)).

17 **Examples.** DSHIFTR (Z'01234567', Z'89ABCDEF', 8) has the value Z'6789ABCD'.
 18 DSHIFTR (B'111', B'000', 2) has the value B'110'. DSHIFTR (I, I, SHIFT) has the same result value
 19 as ISHFTC (I, –SHIFT).

20 **13.7.58 EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])**

21 **Description.** Perform an end-off shift on an array expression of rank one or perform end-off shifts on
 22 all the complete rank-one sections along a given dimension of an array expression of rank two or greater.
 23 Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other
 24 end. Different sections may have different boundary values and may be shifted by different amounts and
 25 in different directions.

26 **Class.** Transformational function.

27 **Arguments.**

28 ARRAY may be of any type. It shall be an array.
 SHIFT shall be of type integer and shall be scalar if ARRAY has rank one; otherwise, it shall
 be scalar or of rank $n - 1$ and of shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$ where
 29 $(d_1, d_2, \dots, d_n)$ 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 may be omitted for the types in the following table 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.

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
Bits	B'0'

DIM (optional) shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. If DIM is omitted, 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 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 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 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 EOSHIFT (V, SHIFT = 3), which has the value [4, 5, 6, 0, 0, 0]; EOSHIFT (V, SHIFT = -2, BOUNDARY = 99) achieves an end-off shift to the right by 2 positions with the boundary value of 99 and has the value [99, 99, 1, 2, 3, 4].

Case (ii): The rows of an array of rank two may all be shifted by the same amount or by different amounts and the boundary elements can be the same or different. If M is the array

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix},$$

then the value of EOSHIFT (M, SHIFT = -1, BOUNDARY = '*', DIM = 2) is

$$\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix},$$

and the value of EOSHIFT (M, SHIFT = [-1, 1, 0], BOUNDARY = ['*', '/', '?'], DIM = 2) is

$$\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}.$$

13.7.59 EPSILON (X)

Description. Returns a positive model number that is almost negligible compared to unity of the model representing numbers of the same type and kind type parameter as the argument.

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 13.4 for the model representing numbers of the same type and kind type parameter as X.

1 **Example.** EPSILON (X) has the value 2^{-23} for real X whose model is as in Note 13.3.

2 **13.7.60 ERF (X)**

3 **Description.** Error function.

4 **Class.** Elemental function.

5 **Argument.** X shall be of type real.

6 **Result Characteristics.** Same as X.

7 **Result Value.** The result has a value equal to a processor-dependent approximation to the error
8 function of X, $\frac{2}{\sqrt{\pi}} \int_0^X \exp(-t^2) dt$.

9 **Example.** ERF (1.0) has the value 0.843 (approximately).

10 **13.7.61 ERFC (X)**

11 **Description.** Complementary error function.

12 **Class.** Elemental function.

13 **Argument.** X shall be of type real.

14 **Result Characteristics.** Same as X.

15 **Result Value.** The result has a value equal to a processor-dependent approximation to the comple-
16 mentary error function of X, that is, $1 - \text{ERF}(X)$.

17 **Example.** ERFC (1.0) has the value 0.157 (approximately).

18 **13.7.62 ERFC_SCALED (X)**

19 **Description.** Exponentially-scaled complementary error function.

20 **Class.** Elemental function.

21 **Argument.** X shall be of type real.

22 **Result Characteristics.** Same as X.

23 **Result Value.** The result has a value equal to a processor-dependent approximation to the exponentially-
24 scaled complementary error function of X, $\exp(X^2) \frac{2}{\sqrt{\pi}} \int_X^\infty \exp(-t^2) dt$.

25 **Example.** ERFC_SCALED (20.0) has the value 0.02817434874 (approximately).

NOTE 13.8

The complementary error function is asymptotic to $\exp(-X^2)/(X\sqrt{\pi})$. As such it underflows for $X > \approx 9$ when using IEEE single precision arithmetic. The exponentially-scaled complementary error function is asymptotic to $1/(X\sqrt{\pi})$. As such it does not underflow until $X > \text{HUGE}(X)/\sqrt{\pi}$.

26 **13.7.63 EXECUTE_COMMAND_LINE (COMMAND [, WAIT, EXITSTAT, CMDSTAT, CMDMSG])**

27 **Description.** Execute the command line specified by the string COMMAND. If the processor supports
28 command line execution, it shall support synchronous and may support asynchronous execution of the

1 command line.

2 **Class.** Subroutine.

3 **Arguments.**

4 COMMAND shall be of type default character and shall be scalar. It is an INTENT(IN) argument. Its value is the command line to be executed. The interpretation is processor-dependent.

5 WAIT (optional) shall be of type default logical and shall be 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.

6 EXITSTAT shall be of type default integer and shall be a scalar. 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.

7 CMDSTAT shall be of type default integer and shall be a scalar. 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.

8 CMDMSG shall be of type default character and shall be a scalar. It is an INTENT(INOUT) argument. If an error condition occurs, it is assigned a processor-dependent explanatory message. Otherwise, it is unchanged.

9 When the command is executed synchronously, EXECUTE_COMMAND_LINE returns after the command line has completed execution. Otherwise, EXECUTE_COMMAND_LINE returns without waiting.

11 If an error condition occurs and CMDSTAT is not present, execution of the program is terminated.

12 **13.7.64 EXP (X)**

13 **Description.** Exponential.

14 **Class.** Elemental function.

15 **Argument.** X shall be of type real or complex.

16 **Result Characteristics.** Same as X.

17 **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.

19 **Example.** EXP (1.0) has the value 2.7182818 (approximately).

20 **13.7.65 EXPONENT (X)**

21 **Description.** Returns the exponent part of the argument when represented as a model number.

22 **Class.** Elemental function.

23 **Argument.** X shall be of type real.

24 **Result Characteristics.** Default integer.

1 **Result Value.** The result has a value equal to the exponent e of the representation for the value of X
 2 in the model (13.4) that has the radix of X but no limits on exponent values, provided X is nonzero and
 3 e is within the range for default integers. If X has the value zero, the result has the value zero. If X is
 4 an IEEE infinity or NaN, the result has the value HUGE(0).

5 **Examples.** EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose
 6 model is as in Note 13.3.

7 **13.7.66 EXTENDS_TYPE_OF (A, MOLD)**

8 **Description.** Inquires whether the dynamic type of A is an extension type (4.5.7) of the dynamic type
 9 of MOLD.

10 **Class.** Inquiry function.

11 **Arguments.**

12 A shall be an object of extensible type. If it is a pointer, it shall not have an undefined
 association status.

13 $MOLD$ shall be an object of extensible type. If it is a pointer, it shall not have an undefined
 association status.

14 **Result Characteristics.** Default logical scalar.

15 **Result Value.** If MOLD is unlimited polymorphic and is either a disassociated pointer or unallocated
 16 allocatable, the result is true; otherwise if A is unlimited polymorphic and is either a disassociated pointer
 17 or unallocated allocatable, the result is false; otherwise the result is true if and only if the dynamic type
 18 of A is an extension type of the dynamic type of MOLD.

NOTE 13.9

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

19 **13.7.67 FINDLOC (ARRAY, VALUE, DIM, [,MASK, KIND, BACK]) or FINDLOC (ARRAY, VALUE, [, MASK, KIND, BACK])**

20 **Description.** Determine the location of the first element of ARRAY identified by MASK along dimen-
 21 sion DIM having a value equal to VALUE.

22 **Class.** Transformational function.

23 **Arguments.**

24 $ARRAY$ shall be of intrinsic type. It shall be an array.

25 $VALUE$ shall be scalar and in type conformance with ARRAY, as specified in Table 7.12.

26 DIM shall be scalar and of type integer with a value in the range $1 \leq DIM \leq n$, where n
 is the rank of ARRAY. The corresponding actual argument shall not be an optional
 dummy argument.

27 $MASK$ (optional) shall be of type logical and shall be conformable with ARRAY.

28 $KIND$ (optional) shall be a scalar integer initialization expression.

29 $BACK$ (optional) shall be scalar and of type logical.

30 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the

1 value of KIND; otherwise the kind type parameter is that of default integer type. If DIM is absent, the
 2 result is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank
 3 $n - 1$ 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.

4 Result Value.

5 *Case (i):* The result of FINDLOC (ARRAY, VALUE) is a rank-one array whose element values are
 6 the values of the subscripts of an element of ARRAY whose value matches VALUE. The
 7 i^{th} subscript returned lies in the range 1 to e_i , where e_i is the extent of the i^{th} dimension
 8 of ARRAY. If no elements match VALUE or ARRAY has size zero, all elements of the
 9 result are zero.

10 *Case (ii):* The result of FINDLOC (ARRAY, VALUE, MASK = MASK) is a rank-one array whose
 11 element values are the values of the subscripts of an element of ARRAY, corresponding
 12 to a true element of MASK, whose value matches VALUE. The i_{th} subscript returned lies
 13 in the range 1 to e_i , where e_i is the extent of the i_{th} dimension of ARRAY. If no elements
 14 match VALUE, ARRAY has size zero, or every element of MASK has the value false, all
 15 elements of the result are zero.

16 *Case (iii):* If ARRAY has rank one, the result of
 17 FINDLOC (ARRAY, VALUE, DIM=DIM [, MASK = MASK]) is a scalar whose value is
 18 equal to that of the first element of FINDLOC (ARRAY, VALUE [, MASK = MASK]).
 19 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
 20 equal to FINDLOC (ARRAY ($s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$), VALUE, DIM=1,
 21 [, MASK = MASK ($s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$))).

22 If both ARRAY and VALUE are of type logical, the comparison is performed with the .EQV. operator;
 23 otherwise, the comparison is performed with the == operator. If the value of the comparison is true,
 24 that element of ARRAY matches VALUE.

25 If only one element matches VALUE, that element's subscripts are returned. Otherwise, if more than
 26 one element matches VALUE and BACK is absent or present with the value false, the element whose
 27 subscripts are returned is the first such element, taken in array element order. If BACK is present with
 28 the value true, the element whose subscripts are returned is the last such element, taken in array element
 29 order.

30 Examples.

31 *Case (i):* The value of FINDLOC ([2, 6, 4, 6], VALUE=6) is [2], and the value of FINDLOC ([2,
 32 6, 4, 6], VALUE=6, BACK=.TRUE.) is [4].

33 *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 $\begin{bmatrix} T & T & F & T \\ T & T & F & T \\ T & T & F & T \end{bmatrix}$, FIND-
 34 LOC (A, 7, MASK = M) has the value [1, 4] and FINDLOC (A, 7, MASK = M, BACK =
 35 .TRUE.) has the value [3, 4]. Note that this is independent of the declared lower bounds
 36 for A.

37 *Case (iii):* The value of FINDLOC ([2, 6, 4], VALUE=6, DIM=1) is 2. If B has the value
 38 $\begin{bmatrix} 1 & 2 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, FINDLOC (B, VALUE=2, DIM=1) has the value [2, 1, 0] and FIND-
 39 LOC (B, VALUE=2, DIM=2) has the value [2, 1]. Note that this is true even if B has
 40 a declared lower bound other than 1.

41 13.7.68 FLOOR (A [, KIND])

42 **Description.** Returns the greatest integer less than or equal to its argument.

43 **Class.** Elemental function.

1 **Arguments.**

2 A shall be of type real.

3 KIND (optional) shall be a scalar integer initialization expression.

4 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
5 value of KIND; otherwise, the kind type parameter is that of default integer type.

6 **Result Value.** The result has a value equal to the greatest integer less than or equal to A.

7 **Examples.** FLOOR (3.7) has the value 3. FLOOR (−3.7) has the value −4.

8 **13.7.69 FORM_TEAM (TEAM, IMAGES [, STAT, ERRMSG])**

9 **Description.** Form a team of images.

10 **Class.** Subroutine.

11 **Arguments.**

12 TEAM shall be a scalar of type IMAGE_TEAM (13.8.2.7). It is an INTENT(OUT) argument.

IMAGES shall be of rank one and type integer. It is an INTENT(IN) argument that specifies
the image indices of the team members, and shall specify the same set of images on
all images of the team. All the elements shall have values in the range 1, ..., NUM_
IMAGES () and there shall be no repeated values. One of the elements shall have the
value THIS_IMAGE().

13 STAT (optional) shall be a scalar of type default integer. It is an INTENT(OUT) argument. It is
assigned a processor-dependent positive value if an error condition occurs, −1 if any
of the images of the team has initiated image termination but no error condition has
occurred, and zero otherwise.

14 ERRMSG shall be a scalar of type default character. It is an INTENT(INOUT) argument. If
(optional) an error condition occurs, it is assigned a processor-dependent explanatory message;
otherwise, it is unchanged.

16 If FORM_TEAM is invoked by an image, it shall be invoked by the same statement on all images specified
17 by the IMAGES argument. There is an implicit team synchronization after the team is formed.

18 If an error condition occurs and STAT is not present, error termination occurs.

19 **Example.**

20 The following code fragment splits images into two groups and implicitly synchronizes each of the
21 teams if there are two or more images. If there is only one image, that image becomes the only
22 team member. The members of the team may be specified in a different order on different images.

```

23     USE, INTRINSIC :: ISO_FORTRAN_ENV
24     INTEGER :: I
25     TYPE(IMAGE_TEAM) :: TEAM
26     IF (THIS_IMAGE() <= NUM_IMAGES()/2) THEN
27         CALL FORM_TEAM(TEAM, [(I, I=1, NUM_IMAGES()/2)])
28     ELSE
29         CALL FORM_TEAM(TEAM, [(I, I=NUM_IMAGES()/2+1, NUM_IMAGES())])
30     END IF

```


1 13.7.70 FRACTION (X)

2 **Description.** Returns the fractional part of the model representation of the argument value.

3 **Class.** Elemental function.

4 **Argument.** X shall be of type real.

5 **Result Characteristics.** Same as X.

6 **Result Value.** The result has the value $X \times b^{-e}$, where b and e are as defined in 13.4 for the representation of X in the model that has the radix of X but no limits on exponent values. If X has the value zero, or is an IEEE infinity or NaN, the result has the same value as X.

9 **Example.** FRACTION (3.0) has the value 0.75 for reals whose model is as in Note 13.3.

10 13.7.71 GAMMA (X)

11 **Description.** Gamma function.

12 **Class.** Elemental function.

13 **Argument.** X shall be of type real. Its value shall not be a negative integer or zero.

14 **Result Characteristics.** Same as X.

15 **Result Value.** The result has a value equal to a processor-dependent approximation to the gamma function of X, $\Gamma(X) = \int_0^\infty t^{X-1} \exp(-t) dt$.

17 **Example.** GAMMA (1.0) has the value 1.000 (approximately).

18 13.7.72 GET_COMMAND ([COMMAND, LENGTH, STATUS])

19 **Description.** Returns the entire command by which the program was invoked.

20 **Class.** Subroutine.

21 **Arguments.**

COMMAND shall be scalar and of type default character. 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 shall be scalar and of type default integer. 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 shall be scalar and of type default integer. 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.

25 13.7.73 GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])

1 **Description.** Returns a command argument.

2 **Class.** Subroutine.

3 **Arguments.**

4 **NUMBER** shall be scalar and of type default integer. It is an INTENT(IN) argument.

5 It specifies the number of the command argument that the other arguments give information about. Useful values of **NUMBER** are those between 0 and the argument count returned by the **COMMAND_ARGUMENT_COUNT** intrinsic. Other values are allowed, but will result in error status return (see below).

6 Command argument 0 is defined to be the command name by which the program was invoked if the processor has such a concept. It is allowed to call the **GET_COMMAND_ARGUMENT** procedure for command argument number 0, even if the processor does not define command names or other command arguments.

7 The remaining command arguments are numbered consecutively from 1 to the argument count in an order determined by the processor.

8 **VALUE** (optional) shall be scalar and of type default character. It is an INTENT(OUT) argument. It is assigned the value of the command argument specified by **NUMBER**. If the command argument value cannot be determined, **VALUE** is assigned all blanks.

9 **LENGTH** (optional) shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the significant length of the command argument specified by **NUMBER**. The significant length may include trailing blanks if the processor allows command arguments with significant 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. If the command argument length cannot be determined, a length of 0 is assigned.

10 **STATUS** (optional) shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the value -1 if the **VALUE** argument is present and has a length less than the significant length of the command argument specified by **NUMBER**. It is assigned a processor-dependent positive value if the argument retrieval fails. Otherwise it is assigned the value 0.

NOTE 13.10

One possible reason for failure is that **NUMBER** is negative or greater than **COMMAND_ARGUMENT_COUNT()**.

11 **Example.**

```

12      Program echo
13      integer :: i
14      character :: command*32, arg*128
15      call get_command_argument(0, command)
16      write (*,*) "Command name is: ", command
17      do i = 1 , command_argument_count()
18          call get_command_argument(i, arg)
19          write (*,*) "Argument ", i, " is ", arg
20      end do

```

1 end program echo

2 **13.7.74 GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])**

3 **Description.** Gets the value of an environment variable.

4 **Class.** Subroutine.

5 **Arguments.**

NAME shall be scalar and of type default character. It is an INTENT(IN) argument. The
6 interpretation of case is processor dependent

VALUE shall be a scalar of type default character. It is an INTENT(OUT) argument. It is
(optional) 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
7 processor does not support environment variables.

LENGTH shall be a scalar of type default integer. It is an INTENT(OUT) argument. If the
(optional) specified environment variable exists and has a value, LENGTH is set to the length
8 of that value. Otherwise LENGTH is set to 0.

STATUS shall be scalar and of type default integer. It is an INTENT(OUT) argument. If the
(optional) environment variable exists and either has no value or its value is successfully assigned
to VALUE, STATUS is set to zero. STATUS is set to -1 if the VALUE argument is
present and has a length less than the significant length of the environment variable.
It is assigned the value 1 if the specified environment variable does not exist, or 2
if the processor does not support environment variables. Processor-dependent values
9 greater than 2 may be returned for other error conditions.

TRIM_NAME shall be a scalar of type logical. It is an INTENT(IN) argument. If TRIM_NAME is
(optional) present with the value false then trailing blanks in NAME are considered significant
if the processor supports trailing blanks in environment variable names. Otherwise
10 trailing blanks in NAME are not considered part of the environment variable's name.

11 **13.7.75 HUGE (X)**

12 **Description.** Returns the largest number of the model representing numbers of the same type and kind
13 type parameter as the argument.

14 **Class.** Inquiry function.

15 **Argument.** X shall be of type integer, real, or bits. It may be a scalar or an array.

16 **Result Characteristics.** Scalar of the same type and kind type parameter as X.

17 **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
18 real, where r , q , b , p , and $e_{\max}$ are as defined in 13.4 for the model representing numbers of the same
19 type and kind type parameter as X. If X is of type bits, the result value has all of its bits set to 1.

20 **Example.** HUGE (X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X whose model is as in Note 13.3.

21 **13.7.76 HYPOT (X, Y)**

22 **Description.** Euclidean distance function.

23 **Class.** Elemental function.

1 **Arguments.**

2 X shall be of type real.

3 Y shall be of type real with the same kind type parameter as X.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to the Euclidean
6 distance, $\sqrt{X^2 + Y^2}$, without undue overflow or underflow.

7 **Example.** HYPOT (2.0, 1.0) has the value 2.236 (approximately).

8 **13.7.77 IACHAR (C [, KIND])**

9 **Description.** Returns the position of a character in the ASCII collating sequence. This is the inverse
10 of the ACHAR function.

11 **Class.** Elemental function.

12 **Arguments.**

13 C shall be of type character and of length one.

14 KIND (optional) shall be a scalar integer initialization expression.

15 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
16 value of KIND; otherwise, the kind type parameter is that of default integer type.

17 **Result Value.** If C is in the collating sequence defined by the codes specified in ISO/IEC 646:1991
18 (International Reference Version), the result is the position of C in that sequence and satisfies the
19 inequality ($0 \leq \text{IACHAR}(C) \leq 127$). A processor-dependent value is returned if C is not in the ASCII
20 collating sequence. The results are consistent with the LGE, LGT, LLE, and LLT lexical comparison
21 functions. For example, if LLE (C, D) is true, $\text{IACHAR}(C) \leq \text{IACHAR}(D)$ is true where C and D are
22 any two characters representable by the processor.

23 **Example.** IACHAR ('X') has the value 88.

24 **13.7.78 IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY, [MASK])**

25 **Description.** Bitwise AND of all the elements of ARRAY along dimension DIM corresponding to the
26 true elements of MASK.

27 **Class.** Transformational function.

28 **Arguments.**

29 ARRAY shall be of type integer or bits. It shall be an array.

DIM shall be a scalar of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n
is the rank of ARRAY. The corresponding actual argument shall not be an optional
dummy argument.

31 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

32 **Result Characteristics.** The result is of the same type and kind type parameter as ARRAY. It is
33 scalar if DIM is absent or if ARRAY has rank one; otherwise, the result is an array of rank $n - 1$ and
34 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.

1 **Result Value.**

2 *Case (i):* If ARRAY has size zero the result value is equal to NOT (INT (0, KIND (ARRAY))) if
 3 ARRAY is of type integer, and NOT (BITS (B'0', KIND (ARRAY))) if array is of type
 4 bits. Otherwise, the result of IALL (ARRAY) has a value equal to the bitwise AND of
 5 all the elements of ARRAY.

6 *Case (ii):* The result of IALL (ARRAY, MASK=MASK) has a value equal to
 7 IALL (PACK (ARRAY, MASK)).

8 *Case (iii):* The result of IALL (ARRAY, DIM=DIM [, MASK=MASK]) has a value equal to that
 9 of IALL (ARRAY [, MASK=MASK]) if ARRAY has rank one. Otherwise, the value of
 10 element ($s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n$) of the result is equal to IALL (ARRAY ($s_1,$
 11 $s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$) [, MASK = MASK ($s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1},$
 12 $\dots, s_n$)]).

13 **Examples.** IALL ([B'1110', B'1101', B'1011']) has the value B'1000'. IALL ([B'1110', B'1101', B'1011'],
 14 MASK=[.true., .false., .true]) has the value B'1010'. IALL([14, 13, 11]) has the value 8. IALL([14, 13, 11],
 15 MASK=[.true., .false., .true]) has the value 10.

16 **13.7.79 IAND (I, J)**

17 **Description.** Performs a bitwise AND.

18 **Class.** Elemental function.

19 **Arguments.**

20 I shall be of type integer or bits.

J shall be of type integer or bits. If both I and J are of type integer, they shall have
 the same kind type parameter; otherwise BITS_KIND (I) shall be equal to BITS-
 21 KIND (J).

22 **Result Characteristics.** If I and J have the same type, the result characteristics are the same as I.
 23 Otherwise, the result characteristics are the same as those of the argument with integer type.

24 **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to the
 25 following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0
0	0	0

26 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

27 **Examples.** IAND (1, 3) has the value 1. IAND (Z'12345678', Z'0000FFFF') has the value Z'00005678'.

28 **13.7.80 IANY (ARRAY, DIM [, MASK]) or IANY (ARRAY, [MASK])**

29 **Description.** Bitwise OR of all the elements of ARRAY along dimension DIM corresponding to the
 30 true elements of MASK.

31 **Class.** Transformational function.

Arguments.

1

2 **ARRAY** shall be of type integer or bits. It shall be an array.3 **DIM** (optional) shall be a scalar and of type integer with a value in the range $1 \leq DIM \leq n$, where n is the rank of **ARRAY**. The corresponding actual argument shall not be an optional dummy argument.4 **MASK** (optional) shall be of type logical and shall be conformable with **ARRAY**.5 **Result Characteristics.** The result is of the same type and kind type parameter as **ARRAY**. It is
6 scalar if **DIM** is absent or if **ARRAY** has rank one; otherwise, the result is an array of rank $n - 1$ and
7 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**.8 **Result Value.**9 *Case (i):* The result of **IANY** (**ARRAY**) is the bitwise OR of all the elements of **ARRAY**. If **AR-**
10 **RAY** has size zero the result value is equal to zero if **ARRAY** is of type integer, and
11 **BITS** (**B'0'**, **KIND**(**ARRAY**)) otherwise.12 *Case (ii):* The result of **IANY** (**ARRAY**, **MASK=MASK**) has a value equal to
13 **IANY** (**PACK** (**ARRAY**, **MASK**)).14 *Case (iii):* The result of **IANY** (**ARRAY**, **DIM=DIM** [, **MASK=MASK**]) has a value equal to that
15 of **IANY** (**ARRAY** [, **MASK=MASK**]) if **ARRAY** has rank one. Otherwise, the value of
16 element $(s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1}, \dots, s_n)$ of the result is equal to **IANY** (**ARRAY** ($s_1,$
17 $s_2, \dots, s_{DIM-1}, :, s_{DIM+1}, \dots, s_n$) [, **MASK** = **MASK**($s_1, s_2, \dots, s_{DIM-1}, :, s_{DIM+1}, \dots,$
18 s_n)]).19 **Examples.** **IANY** ([**B'1110'**, **B'1101'**, **B'1000'**]) has the value **B'1111'**. **IANY** ([**B'1110'**, **B'1101'**,
20 **B'1000'**], **MASK**=[**true.**, **false.**, **true.**]) has the value **B'1110'**. **IANY** ([14, 13, 8]) has the value 15.
21 **IANY** ([14, 13, 8], **MASK**=[**true.**, **false.**, **true.**]) has the value 14.22 **13.7.81 IBCLR (I, POS)**23 **Description.** Clears one bit to zero.24 **Class.** Elemental function.25 **Arguments.**26 **I** shall be of type integer or bits.27 **POS** shall be of type integer. It shall be nonnegative and less than **BIT.SIZE** (**I**).28 **Result Characteristics.** Same as **I**.29 **Result Value.** The result has the value of the sequence of bits of **I**, except that bit **POS** is zero. The
30 model for the interpretation of an integer value as a sequence of bits is in 13.3.31 **Examples.** **IBCLR** (14, 1) has the value 12. If **V** has the value [1, 2, 3, 4], the value of **IB-**
32 **CLR** (**POS** = **V**, **I** = 31) is [29, 27, 23, 15]. **IBCLR** (**B'11111'**, 3) has the value **B'10111'**.33 **13.7.82 IBITS (I, POS, LEN)**34 **Description.** Extracts a sequence of bits.35 **Class.** Elemental function.36 **Arguments.**

1 I shall be of type integer or bits.
 2 POS shall be of type integer. It shall be nonnegative and POS + LEN shall be less than
 3 or equal to BIT_SIZE (I).
 4 LEN shall be of type integer and nonnegative.

5 **Result Characteristics.** Same as I.

6 **Result Value.** The result has the value of the sequence of LEN bits in I beginning at bit POS, right-
 7 adjusted and with all other bits zero. The model for the interpretation of an integer value as a sequence
 8 of bits is in 13.3.

9 **Examples.** IBITS (14, 1, 3) has the value 7. IBITS (Z'ABCD', 4, 8) has the value Z'00BC'.

10 13.7.83 IBSET (I, POS)

11 **Description.** Sets one bit to one.

12 **Class.** Elemental function.

13 **Arguments.**

14 I shall be of type integer or bits.
 15 POS shall be of type integer. It shall be nonnegative and less than BIT_SIZE (I).

16 **Result Characteristics.** Same as I.

17 **Result Value.** The result has the value of the sequence of bits of I, except that bit POS is one. The
 18 model for the interpretation of an integer value as a sequence of bits is in 13.3.

19 **Examples.** IBSET (12, 1) has the value 14. If V has the value [1, 2, 3, 4], the value of IBSET (POS = V,
 20 I = 0) is [2, 4, 8, 16]. IBSET (B'00000', 3) has the value B'01000'.

21 13.7.84 ICHAR (C [, KIND])

22 **Description.** Returns the position of a character in the processor collating sequence associated with
 23 the kind type parameter of the character. This is the inverse of the CHAR function.

24 **Class.** Elemental function.

25 **Arguments.**

26 C shall be of type character and of length one. Its value shall be that of a character
 27 capable of representation in the processor.
 28 KIND (optional) shall be a scalar integer initialization expression.

29 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
 30 value of KIND; otherwise, the kind type parameter is that of default integer type.

31 **Result Value.** The result is the position of C in the processor collating sequence associated with
 32 the kind type parameter of C and is in the range $0 \leq \text{ICHAR}(C) \leq n - 1$, where n is the number
 33 of characters in the collating sequence. For any characters C and D capable of representation in the
 34 processor, C <= D is true if and only if ICHAR (C) <= ICHAR (D) is true and C == D is true if and
 only if ICHAR (C) == ICHAR (D) is true.

35 **Example.** ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence for the

1 default character type.

2 **13.7.85 IEOR (I, J)**

3 **Description.** Performs a bitwise exclusive OR.

4 **Class.** Elemental function.

5 **Arguments.**

6 I shall be of type integer or bits.

J shall be of type integer or bits. If both I and J are of type integer, they shall have the same kind type parameter; otherwise BITS_KIND (I) shall be equal to BITS_KIND (J).

8 **Result Characteristics.** If I and J have the same type, the result characteristics are the same as I.
9 Otherwise, the result characteristics are the same as those of the argument with integer type.

10 **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to the
11 following truth table:

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

12 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

13 **Examples.** IEOR (1, 3) has the value 2. IEOR (O'5', B'110') has the value O'3'.

14 **13.7.86 IMAGE_INDEX (CO_ARRAY, SUB)**

15 **Description.** Returns the index of the image corresponding to the co-subscripts SUB for CO_ARRAY.

16 **Class.** Inquiry function.

17 **Arguments.**

18 CO_ARRAY shall be a co-array of any type.

SUB shall be a rank-one array of size equal to the co-rank of CO_ARRAY and shall be of type integer.

20 **Result Characteristics.** Default integer scalar.

21 **Result Value.** If the value of SUB is a valid sequence of co-subscripts for CO_ARRAY, the result is
22 the index of the corresponding image. Otherwise, the result is zero.

23 **Examples.** If A is declared A [0:*], IMAGE_INDEX (A, [0]) has the value 1. If B is declared REAL
24 B (10, 20) [10, 0:9, 0:*], IMAGE_INDEX (B, [3, 1, 2]) has the value 213 (on any image).

NOTE 13.11

For an example of a module that implements a function similar to the intrinsic IMAGE_INDEX, see subclause C.10.1.

1 13.7.87 INDEX (STRING, SUBSTRING [, BACK, KIND])

2 **Description.** Returns the starting position of a substring within a string.

3 **Class.** Elemental function.

4 **Arguments.**

5 STRING shall be of type character.

6 SUBSTRING shall be of type character with the same kind type parameter as STRING.

7 BACK (optional) shall be of type logical.

8 KIND (optional) shall be a scalar integer initialization expression.

9 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
10 value of KIND; otherwise the kind type parameter is that of default integer type.

11 **Result Value.**

12 *Case (i):* If BACK is absent or has the value false, the result is the minimum positive value of I
13 such that $\text{STRING}(\text{I} : \text{I} + \text{LEN}(\text{SUBSTRING}) - 1) = \text{SUBSTRING}$ or zero if there
14 is no such value. Zero is returned if $\text{LEN}(\text{STRING}) < \text{LEN}(\text{SUBSTRING})$ and one is
15 returned if $\text{LEN}(\text{SUBSTRING}) = 0$.

16 *Case (ii):* If BACK is present with the value true, the result is the maximum value of I less than or
17 equal to $\text{LEN}(\text{STRING}) - \text{LEN}(\text{SUBSTRING}) + 1$ such that $\text{STRING}(\text{I} : \text{I} + \text{LEN}(\text{SUB-}$
18 $\text{STRING}) - 1) = \text{SUBSTRING}$ or zero if there is no such value. Zero is returned if
19 $\text{LEN}(\text{STRING}) < \text{LEN}(\text{SUBSTRING})$ and $\text{LEN}(\text{STRING}) + 1$ is returned if $\text{LEN}(\text{SUB-}$
20 $\text{STRING}) = 0$.

21 **Examples.** INDEX ('FORTRAN', 'R') has the value 3.

22 INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

23 13.7.88 INT (A [, KIND])

24 **Description.** Convert to integer type.

25 **Class.** Elemental function.

26 **Arguments.**

27 A shall be of type integer, real, complex, or bits.

28 KIND (optional) shall be a scalar integer initialization expression.

29 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
30 value of KIND; otherwise, the kind type parameter is that of default integer type.

31 **Result Value.**

32 *Case (i):* If A is of type integer, $\text{INT}(A) = A$.

33 *Case (ii):* If A is of type real, there are two cases: if $|A| < 1$, INT (A) has the value 0; if $|A| \geq 1$,
34 INT (A) is the integer whose magnitude is the largest integer that does not exceed the
35 magnitude of A and whose sign is the same as the sign of A.

36 *Case (iii):* If A is of type complex, $\text{INT}(A) = \text{INT}(\text{REAL}(A, \text{KIND}(A)))$.

37 *Case (iv):* If A is of type bits, the result has the integer value such that
38 $\text{BITS}(\text{INT}(A, \text{kind})) = \text{BITS}(A, \text{BITS\_KIND}(\text{INT}(0, \text{kind})))$, where *kind* is the kind of

1 the result. If the value specified by the model in 13.3 for interpreting the bits as an integer
2 is a valid value for the result, it has that value; otherwise, it is processor dependent.

3 **Example.** INT (−3.7) has the value −3.

4 **13.7.89 IOR (I, J)**

5 **Description.** Performs a bitwise inclusive OR.

6 **Class.** Elemental function.

7 **Arguments.**

8 I shall be of type integer or bits.

J shall be of type integer or bits. If both I and J are of type integer, they shall have
the same kind type parameter; otherwise BITS_KIND (I) shall be equal to BITS-
9 KIND (J).

10 **Result Characteristics.** If I and J have the same type, the result characteristics are the same as I.
11 Otherwise, the result characteristics are the same as those of the argument with integer type.

12 **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to the
13 following truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

14 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

15 **Examples.** IOR (5, 3) has the value 7. IOR (Z'1234, Z'00FF') has the value Z'12FF'.

16 **13.7.90 IPARITY (ARRAY, DIM [, MASK]) or IPARITY (ARRAY, [MASK])**

17 **Description.** Bitwise exclusive OR of all the elements of ARRAY along dimension DIM corresponding
18 to the true elements of MASK.

19 **Class.** Transformational function.

20 **Arguments.**

21 ARRAY shall be of type integer or bits. It shall be an array.

DIM shall be a scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n
is the rank of ARRAY. The corresponding actual argument shall not be an optional
22 dummy argument.

23 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

24 **Result Characteristics.** The result is of the same type and kind type parameter as ARRAY. It is
25 scalar if DIM is absent; otherwise, the result has rank $n - 1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$
26 $\dots, d_n)$ where $(d_1, d_2, \dots, d_n)$ is the shape of ARRAY.

27 **Result Value.**

- 1 *Case (i):* The result of IPARITY (ARRAY) has a value equal to the bitwise exclusive OR of all the
 2 elements of ARRAY. If ARRAY has size zero the result has the value zero if ARRAY is
 3 of type integer, and BITS (B'0', KIND (ARRAY)) otherwise.
- 4 *Case (ii):* The result of IPARITY (ARRAY, MASK=MASK) has a value equal to that of IPAR-
 5 ITY (PACK (ARRAY, MASK)).
- 6 *Case (iii):* The result of IPARITY (ARRAY, DIM=DIM [, MASK=MASK]) has a value equal to
 7 that of IPARITY (ARRAY [, MASK=MASK]) if ARRAY has rank one. Otherwise,
 8 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
 9 IPARITY (ARRAY $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ [, MASK = MASK($s_1, s_2,$
 10 $\dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$)).
- 11 **Examples.** IPARITY ([B'1110', B'1101', B'1000']) has the value B'1011'.
 12 IPARITY ([B'1110', B'1101', B'1000'], MASK=[.true., .false., .true]) has the value B'0110'.
 13 IPARITY ([14, 13, 8]) has the value 11. IPARITY ([14, 13, 8], MASK=[.true., .false., .true]) has the
 14 value 6.

15 13.7.91 ISHFT (I, SHIFT)

16 **Description.** Performs a logical shift.

17 **Class.** Elemental function.

18 **Arguments.**

- 19 I shall be of type integer or bits.
- SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or equal to
 20 BIT_SIZE (I).

21 **Result Characteristics.** Same as I.

22 **Result Value.** The result has the value obtained by shifting the bits of I by SHIFT positions. If SHIFT
 23 is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no
 24 shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are
 25 shifted in from the opposite end. The model for the interpretation of an integer value as a sequence of
 26 bits is in 13.3.

27 **Examples.** ISHFT (3, 1) has the value 6. ISHFT (B'00000011', 1) has the value B'00000110'.

28 13.7.92 ISHFTC (I, SHIFT [, SIZE])

29 **Description.** Performs a circular shift of the rightmost bits.

30 **Class.** Elemental function.

31 **Arguments.**

- 32 I shall be of type integer or bits.
- SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or equal to
 33 SIZE.
- SIZE (optional) shall be of type integer. The value of SIZE shall be positive and shall not exceed
 34 BIT_SIZE (I). If SIZE is absent, it is as if it were present with the value of BIT-
 SIZE (I).

35 **Result Characteristics.** Same as I.

1 **Result Value.** The result has the value obtained by shifting the SIZE rightmost bits of I circularly by
2 SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the
3 right; and if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered.
4 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

5 **Examples.** ISHFTC (3, 2, 3) has the value 5. ISHFTC (Z'ABCD', 4, 8) has the value Z'ABDC'.

6 **13.7.93 IS_CONTIGUOUS (A)**

7 **Description.** Determine whether an object is contiguous (5.3.6).

8 **Class.** Inquiry function.

9 **Argument.** A may be of any type. It shall be an array. If it is a pointer it shall be associated.

10 **Result Characteristics.** Default logical scalar.

11 **Result Value.** The result has the value true if A is contiguous, and false otherwise.

12 **Example.** After the pointer assignment AP => TARGET (1:10:2), IS_CONTIGUOUS (AP) will have
13 the value false.

14 **13.7.94 IS_IOSTAT_END (I)**

15 **Description.** Determine whether a value indicates an end-of-file condition.

16 **Class.** Elemental function.

17 **Argument.** I shall be of type integer.

18 **Result Characteristics.** Default logical.

19 **Result Value.** The result has the value true if and only if I is a value for the *scalar-int-variable* in an
20 IOSTAT= specifier (9.10.4) that would indicate an end-of-file condition.

21 **13.7.95 IS_IOSTAT_EOR (I)**

22 **Description.** Determine whether a value indicates an end-of-record condition.

23 **Class.** Elemental function.

24 **Argument.** I shall be of type integer.

25 **Result Characteristics.** Default logical.

26 **Result Value.** The result has the value true if and only if I is a value for the *scalar-int-variable* in an
27 IOSTAT= specifier (9.10.4) that would indicate an end-of-record condition.

28 **13.7.96 KIND (X)**

29 **Description.** Returns the value of the kind type parameter of X.

30 **Class.** Inquiry function.

31 **Argument.** X may be of any intrinsic type. It may be a scalar or an array.

32 **Result Characteristics.** Default integer scalar.

33 **Result Value.** The result has a value equal to the kind type parameter value of X.

1 **Example.** KIND (0.0) has the kind type parameter value of default real.

2 **13.7.97 LBOUND (ARRAY [, DIM, KIND])**

3 **Description.** Returns all the lower bounds or a specified lower bound of an array.

4 **Class.** Inquiry function.

5 **Arguments.**

6 ARRAY may be of any type. It shall be an array. It shall not be an unallocated allocatable or a pointer that is not associated.

7 DIM (optional) shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.

8 KIND (optional) shall be a scalar integer initialization expression.

9 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
10 value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if
11 DIM is present; otherwise, the result is an array of rank one and size n , where n is the rank of ARRAY.

12 **Result Value.**

13 *Case (i):* If ARRAY is a whole array or array structure component and either ARRAY is an
14 assumed-size array of rank DIM or dimension DIM of ARRAY has nonzero extent,
15 LBOUND (ARRAY, DIM) has a value equal to the lower bound for subscript DIM of
16 ARRAY. Otherwise the result value is 1.

17 *Case (ii):* LBOUND (ARRAY) has a value whose i th element is equal to LBOUND (ARRAY, i),
18 for $i = 1, 2, \dots, n$, where n is the rank of ARRAY.

19 **Examples.** If A is declared by the statement

20 REAL A (2:3, 7:10)

21 then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

22 **13.7.98 LEADZ (I)**

23 **Description.** Count the number of leading zero bits.

24 **Class.** Elemental function.

25 **Argument.** I shall be of type integer or bits.

26 **Result Characteristics.** Default integer.

27 **Result Value.** If all of the bits of I are zero, the result has the value BIT_SIZE (I). Otherwise, the
28 result has the value $\text{BIT\_SIZE (I)} - 1 - k$, where k is the position of the leftmost 1 bit in I. The model
29 for the interpretation of an integer value as a sequence of bits is in 13.3.

30 **Examples.** LEADZ (B'001101000') has the value 2. LEADZ (1) has the value 31 if BIT_SIZE (1) has
31 the value 32.

32 **13.7.99 LEN (STRING [, KIND])**

33 **Description.** Returns the length of a character entity.

1 **Class.** Inquiry function.

2 **Arguments.**

3 STRING shall be of type character. It may be a scalar or an array. If it is an unallocated
4 allocatable or a pointer that is not associated, its length type parameter shall not be
5 deferred.

6 KIND (optional) shall be a scalar integer initialization expression.

7 **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that specified
8 by the value of KIND; otherwise the kind type parameter is that of default integer type.

9 **Result Value.** The result has a value equal to the number of characters in STRING if it is scalar or in
10 an element of STRING if it is an array.

11 **Example.** If C is declared by the statement

12 CHARACTER (11) C (100)

13 LEN (C) has the value 11.

12 **13.7.100 LEN_TRIM (STRING [, KIND])**

13 **Description.** Returns the length of the character argument without counting trailing blank characters.

14 **Class.** Elemental function.

15 **Arguments.**

16 STRING shall be of type character.

17 KIND (optional) shall be a scalar integer initialization expression.

18 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
19 value of KIND; otherwise the kind type parameter is that of default integer type.

20 **Result Value.** The result has a value equal to the number of characters remaining after any trailing
21 blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

22 **Examples.** LEN_TRIM (' A B ') has the value 4 and LEN_TRIM (' ') has the value 0.

23 **13.7.101 LGE (STRING_A, STRING_B)**

24 **Description.** Test whether a string is lexically greater than or equal to another string, based on the
25 ASCII collating sequence.

26 **Class.** Elemental function.

27 **Arguments.**

28 STRING_A shall be of type default character of type ASCII character.

29 STRING_B shall be of type character with the same kind type parameter as STRING_A.

30 **Result Characteristics.** Default logical.

31 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were
32 extended on the right with blanks to the length of the longer string. If either string contains a character
33 not in the ASCII character set, the result is processor dependent. The result is true if the strings are

1 equal or if `STRING_A` follows `STRING_B` in the ASCII collating sequence; otherwise, the result is false.

NOTE 13.12

The result is true if both `STRING_A` and `STRING_B` are of zero length.

2 **Example.** `LGE ('ONE', 'TWO')` has the value false.

3 **13.7.102 LGT (`STRING_A`, `STRING_B`)**

4 **Description.** Test whether a string is lexically greater than another string, based on the ASCII collating
5 sequence.

6 **Class.** Elemental function.

7 **Arguments.**

8 `STRING_A` shall be of type default character of type ASCII character.

9 `STRING_B` shall be of type character with the same kind type parameter as `STRING_A`.

10 **Result Characteristics.** Default logical.

11 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were
12 extended on the right with blanks to the length of the longer string. If either string contains a character
13 not in the ASCII character set, the result is processor dependent. The result is true if `STRING_A` follows
14 `STRING_B` in the ASCII collating sequence; otherwise, the result is false.

NOTE 13.13

The result is false if both `STRING_A` and `STRING_B` are of zero length.

15 **Example.** `LGT ('ONE', 'TWO')` has the value false.

16 **13.7.103 LLE (`STRING_A`, `STRING_B`)**

17 **Description.** Test whether a string is lexically less than or equal to another string, based on the ASCII
18 collating sequence.

19 **Class.** Elemental function.

20 **Arguments.**

21 `STRING_A` shall be of type default character of type ASCII character.

22 `STRING_B` shall be of type character with the same kind type parameter as `STRING_A`.

23 **Result Characteristics.** Default logical.

24 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were
25 extended on the right with blanks to the length of the longer string. If either string contains a character
26 not in the ASCII character set, the result is processor dependent. The result is true if the strings are
27 equal or if `STRING_A` precedes `STRING_B` in the ASCII collating sequence; otherwise, the result is
28 false.

NOTE 13.14

The result is true if both `STRING_A` and `STRING_B` are of zero length.

1 **Example.** LLE ('ONE', 'TWO') has the value true.

2 **13.7.104 LLT (STRING_A, STRING_B)**

3 **Description.** Test whether a string is lexically less than another string, based on the .

4 **Class.** Elemental function.

5 **Arguments.**

6 STRING_A shall be of type default character of type ASCII character.

7 STRING_B shall be of type character with the same kind type parameter as STRING_A.

8 **Result Characteristics.** Default logical.

9 **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
 11 not in the ASCII character set, the result is processor dependent. The result is true if STRING_A
 12 precedes STRING_B in the ASCII collating sequence; otherwise, the result is false.

NOTE 13.15

The result is false if both STRING_A and STRING_B are of zero length.

13 **Example.** LLT ('ONE', 'TWO') has the value true.

14 **13.7.105 LOG (X)**

15 **Description.** Natural logarithm.

16 **Class.** Elemental function.

17 **Argument.** X shall be of type real or complex. If X is real, its value shall be greater than zero. If X is
 18 complex, its value shall not be zero.

19 **Result Characteristics.** Same as X.

20 **Result Value.** The result has a value equal to a processor-dependent approximation to $\log_e X$. A result
 21 of type complex is the principal value with imaginary part ω in the range $-\pi \leq \omega \leq \pi$. If the real
 22 part of X is less than zero and the imaginary part of X is zero, then the imaginary part of the result
 23 is approximately π if the imaginary part of X is positive real zero or the processor cannot distinguish
 24 between positive and negative real zero, and approximately $-\pi$ if the imaginary part of X is negative
 25 real zero.

26 **Example.** LOG (10.0) has the value 2.3025851 (approximately).

27 **13.7.106 LOG_GAMMA (X)**

28 **Description.** Logarithm of the absolute value of the gamma function.

29 **Class.** Elemental function.

30 **Argument.** X shall be of type real. Its value shall not be a negative integer or zero.

31 **Result Characteristics.** Same as X.

32 **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).

13.7.107 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).

13.7.108 LOGICAL (L [, KIND])

Description. Converts between kinds of logical or from bits to logical.

Class. Elemental function.

Arguments.

L shall be of type logical or bits.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Logical. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default logical.

Result Value.

Case (i): If L is of type logical, the value is that of L.

Case (ii): If L is of type bits and KIND (L) is greater than or equal to BITS_KIND (*result*), the physical representation of the result is the same as that of the rightmost bits of L.

Case (iii): If L is of type bits and KIND (L) is less than BITS_KIND (*result*), the rightmost KIND(L) bits of the physical representation of the result are the same as those of L, and the remaining bits of the physical representation of the result are zero.

NOTE 13.16

The result of a bits to logical conversion may be used in a context requiring a logical value only if the physical representation of the result is valid as a logical value.

Examples. LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical, regardless of the kind type parameter of the logical variable L. LOGICAL (BITS (.TRUE.)) has the value true.

13.7.109 MASKL (I [, KIND])

Description. Generate a left justified mask.

Class. Elemental function.

Arguments.

I shall be of type integer. It shall be nonnegative and less than or equal to the kind type parameter of the result.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Bits. 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 bits type.

Result Value. The result value has its leftmost I bits set to 1 and the remaining bits set to 0.

Example. MASKL (12) has the value Z'FFF0000' if the default bits kind type parameter value is 32.

13.7.110 MASKR (I [, KIND])

Description. Generate a right justified mask.

Class. Elemental function.

Arguments.

I shall be of type integer. It shall be nonnegative and less than or equal to the kind type parameter of the result.

KIND (optional) shall be a scalar integer initialization expression.

Result Characteristics. Bits. 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 bits type.

Result Value. The result value has its rightmost I bits set to 1 and the remaining bits set to 0.

Example. MASKR (12) has the value Z'00000FFF' if the default bits kind type parameter value is 32.

13.7.111 MATMUL (MATRIX_A, MATRIX_B)

Description. Performs matrix multiplication of numeric or logical matrices.

Class. Transformational function.

Arguments.

MATRIX_A shall be of numeric type (integer, real, or complex) or of logical type. It shall be an array of rank one or two.

MATRIX_B shall be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It shall be an array of rank one or two. If MATRIX_A has rank one, MATRIX_B shall have rank two. If MATRIX_B has rank one, MATRIX_A shall have rank two. The size of the first (or only) dimension of MATRIX_B shall equal the size of the last (or only) dimension of MATRIX_A.

Result Characteristics. If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of the arguments as specified in 7.1.9.2 for the * operator. If the arguments are of type logical, the result is of type logical with the kind type parameter of the arguments as specified in 7.1.9.2 for the .AND. operator. The shape of the result depends on the shapes of the arguments as follows:

Case (i): If MATRIX_A has shape (n, m) and MATRIX_B has shape (m, k) , the result has shape (n, k) .

Case (ii): If MATRIX_A has shape (m) and MATRIX_B has shape (m, k) , the result has shape (k) .

Case (iii): If MATRIX_A has shape (n, m) and MATRIX_B has shape (m) , the result has shape (n) .

1 **Result Value.**

2 *Case (i):* Element (i, j) of the result has the value $\text{SUM}(\text{MATRIX\_A}(i, :) * \text{MATRIX\_B}(:, j))$
 3 if the arguments are of numeric type and has the value $\text{ANY}(\text{MATRIX\_A}(i, :) \text{ .AND.}$
 4 $\text{MATRIX\_B}(:, j))$ if the arguments are of logical type.

5 *Case (ii):* Element (j) of the result has the value $\text{SUM}(\text{MATRIX\_A}(:) * \text{MATRIX\_B}(:, j))$ if the
 6 arguments are of numeric type and has the value $\text{ANY}(\text{MATRIX\_A}(:) \text{ .AND.}$ $\text{MATRIX\_B}(:, j))$ if the arguments are of logical type.

7
 8 *Case (iii):* Element (i) of the result has the value $\text{SUM}(\text{MATRIX\_A}(i, :) * \text{MATRIX\_B}(:))$ if
 9 the arguments are of numeric type and has the value $\text{ANY}(\text{MATRIX\_A}(i, :) \text{ .AND.}$
 10 $\text{MATRIX\_B}(:))$ if the arguments are of logical type.

11 **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]
 12 and [1, 2, 3].

13 *Case (i):* The result of MATMUL (A, B) is the matrix-matrix product AB with the value $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$.

14 *Case (ii):* The result of MATMUL (X, A) is the vector-matrix product XA with the value [5, 8, 11].

15 *Case (iii):* The result of MATMUL (A, Y) is the matrix-vector product AY with the value [14, 20].

16 **13.7.112 MAX (A1, A2 [, A3, ...])**

17 **Description.** Maximum value.

18 **Class.** Elemental function.

19 **Arguments.** The arguments shall all have the same type which shall be integer, real, bits, or character
 20 and they shall all have the same kind type parameter.

21 **Result Characteristics.** The type and kind type parameter of the result are the same as those of
 22 the arguments. For arguments of character type, the length of the result is the length of the longest
 23 argument.

24 **Result Value.** The value of the result is that of the largest argument. For arguments of character
 25 type, the result is the value that would be selected by application of intrinsic relational operators; that
 26 is, the collating sequence for characters with the kind type parameter of the arguments is applied. If the
 27 selected argument is shorter than the longest argument, the result is extended with blanks on the right
 28 to the length of the longest argument.

29 **Examples.** MAX (-9.0, 7.0, 2.0) has the value 7.0, MAX ('Z', 'BB') has the value 'Z ', and MAX (['A', 'Z'],
 30 ['BB', 'Y ']) has the value ['BB', 'Z ']. MAX (B'10000', B'01111') has the value B'10000'.

31 **13.7.113 MAXEXPONENT (X)**

32 **Description.** Returns the maximum exponent of the model representing numbers of the same type and
 33 kind type parameter as the argument.

34 **Class.** Inquiry function.

35 **Argument.** X shall be of type real. It may be a scalar or an array.

36 **Result Characteristics.** Default integer scalar.

37 **Result Value.** The result has the value e_{max} , as defined in 13.4 for the model representing numbers of
 the same type and kind type parameter as X.

Example. MAXEXPONENT (X) has the value 127 for real X whose model is as in Note 13.3.

13.7.114 MAXLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MAXLOC (ARRAY [, MASK, KIND, BACK])

Description. Determine the location of the first element of ARRAY along dimension DIM having the maximum value of the elements identified by MASK.

Class. Transformational function.

Arguments.

ARRAY shall be of type integer, real, bits, or character. It shall be an array.

DIM shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.

MASK (optional) shall be of type logical and shall be conformable with ARRAY.

KIND (optional) shall be a scalar integer initialization expression.

BACK (optional) shall be scalar and of type logical.

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 is absent, 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): The result of MAXLOC (ARRAY) 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): The result of MAXLOC (ARRAY, MASK = MASK) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value 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, MAXLOC (ARRAY, DIM = DIM [, MASK = MASK]) is a scalar whose value is equal to that of the first element of MAXLOC (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 the result is equal to

$$\text{MAXLOC (ARRAY (} s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n), \text{DIM}=1 \\ [, \text{MASK} = \text{MASK (} s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

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.

1 If ARRAY has type character, the result is the value that would be selected by application of intrinsic
 2 relational operators; that is, the collating sequence for characters with the kind type parameter of the
 3 arguments is applied.

4 Examples.

5 *Case (i):* The value of MAXLOC ([2, 6, 4, 6]) is [2] and the value of MAXLOC ([2, 6, 4, 6],
 6 BACK=.TRUE.) is [4].

7 *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
 8 [3, 2]. Note that this is independent of the declared lower bounds for A.

9 *Case (iii):* The value of MAXLOC ([5, -9, 3], DIM = 1) is 1. If B has the value $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$,
 10 MAXLOC (B, DIM = 1) is [2, 1, 2] and MAXLOC (B, DIM = 2) is [2, 3]. Note that this
 11 is independent of the declared lower bounds for B.

12 13.7.115 MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])

13 **Description.** Maximum value of the elements of ARRAY along dimension DIM corresponding to the
 14 true elements of MASK.

15 **Class.** Transformational function.

16 Arguments.

17 ARRAY shall be of type integer, real, bits, or character. It shall be an array.

DIM shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n
 is the rank of ARRAY. The corresponding actual argument shall not be an optional
 18 dummy argument.

19 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

20 **Result Characteristics.** The result is of the same type and type parameters as ARRAY. It is scalar
 21 if DIM is absent; otherwise, the result has rank $n - 1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$
 22 where $(d_1, d_2, \dots, d_n)$ is the shape of ARRAY.

23 Result Value.

24 *Case (i):* The result of MAXVAL (ARRAY) has a value equal to the maximum value of all the
 25 elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size zero and type
 26 integer or real, the result has the value of the negative number of the largest magnitude
 27 supported by the processor for numbers of the type and kind type parameter of ARRAY. If
 28 ARRAY has size zero and type character, the result has the value of a string of characters
 29 of length LEN (ARRAY), with each character equal to CHAR (0, KIND (ARRAY)). If
 30 ARRAY has size zero and type bits, the result has the value BITS (B'0', KIND(ARRAY)).

31 *Case (ii):* The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to that of MAX-
 32 VAL (PACK (ARRAY, MASK)).

33 *Case (iii):* The result of MAXVAL (ARRAY, DIM = DIM [,MASK = MASK]) has a value equal to
 34 that of MAXVAL (ARRAY [,MASK = MASK]) if ARRAY has rank one. Otherwise, the
 35 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

$$\text{MAXVAL} (\text{ARRAY} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) \\ [, \text{MASK} = \text{MASK} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

38 If ARRAY has type character, the result is the value that would be selected by application of intrinsic

1 relational operators; that is, the collating sequence for characters with the kind type parameter of the
2 arguments is applied.

3 **Examples.**

4 *Case (i):* The value of MAXVAL ([1, 2, 3]) is 3.

5 *Case (ii):* MAXVAL (C, MASK = C < 0.0) is the maximum of the negative elements of C.

6 *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,
7 DIM = 2) is [5, 7].

8 **13.7.116 MERGE (TSOURCE, FSOURCE, MASK)**

9 **Description.** Choose alternative value according to the value of a mask.

10 **Class.** Elemental function.

11 **Arguments.**

12 TSOURCE may be of any type.

13 FSOURCE shall be of the same type and type parameters as TSOURCE.

14 MASK shall be of type logical.

15 **Result Characteristics.** Same as TSOURCE.

16 **Result Value.** The result is TSOURCE if MASK is true and FSOURCE otherwise.

17 **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
18 the array $\begin{bmatrix} T & . & T \\ . & . & T \end{bmatrix}$, where “T” represents true and “.” represents false, then MERGE (TSOURCE,
19 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
20 for K = -2.

21 **13.7.117 MERGE_BITS (I, J, MASK)**

22 **Description.** Merge bits under mask.

23 **Class.** Elemental function.

24 **Arguments.**

25 I shall be of type bits or integer.

26 J shall be of type integer or bits. BITS_KIND (I) shall be equal to BITS_KIND (J).

27 MASK shall be of type integer or bits. BITS_KIND (I) shall be equal to BITS_KIND (MASK).

28 All integer arguments shall have the same kind type parameter.

29 **Result Characteristics.** If I, J and MASK all have the same type, the result characteristics are the
30 same as I. Otherwise, the result characteristics are the same as those of the argument(s) with integer
31 type.

32 **Result Value.** The result has the value of IOR (IAND (I, MASK), IAND (J, NOT (MASK))).

1 **Example.** MERGE_BITS(Z'0123', Z'89AB', Z'FF00') has the value Z'01AB'.

2 **13.7.118 MIN (A1, A2 [, A3, ...])**

3 **Description.** Minimum value.

4 **Class.** Elemental function.

5 **Arguments.** The arguments shall all be of the same type which shall be integer, real, bits, or character
6 and they shall all have the same kind type parameter.

7 **Result Characteristics.** The type and kind type parameter of the result are the same as those of
8 the arguments. For arguments of character type, the length of the result is the length of the longest
9 argument.

10 **Result Value.** The value of the result is that of the smallest argument. For arguments of character
11 type, the result is the value that would be selected by application of intrinsic relational operators; that
12 is, the collating sequence for characters with the kind type parameter of the arguments is applied. If the
13 selected argument is shorter than the longest argument, the result is extended with blanks on the right
14 to the length of the longest argument.

15 **Examples.** MIN (-9.0, 7.0, 2.0) has the value -9.0, MIN ('A', 'YY') has the value 'A ', and
16 MIN (['Z', 'A'], ['YY', 'B ']) has the value ['YY', 'A ']. MIN (B'10000', B'01111') has the value B'01111'.

17 **13.7.119 MINEXPONENT (X)**

18 **Description.** Returns the minimum (most negative) exponent of the model representing numbers of
19 the same type and kind type parameter as the argument.

20 **Class.** Inquiry function.

21 **Argument.** X shall be of type real. It may be a scalar or an array.

22 **Result Characteristics.** Default integer scalar.

23 **Result Value.** The result has the value $e_{\min}$, as defined in 13.4 for the model representing numbers of
24 the same type and kind type parameter as X.

25 **Example.** MINEXPONENT (X) has the value -126 for real X whose model is as in Note 13.3.

26 **13.7.120 MINLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MINLOC (ARRAY [, MASK, KIND, BACK])**

27 **Description.** Determine the location of the first element of ARRAY along dimension DIM having the
28 minimum value of the elements identified by MASK.

29 **Class.** Transformational function.

30 **Arguments.**

31 ARRAY shall be of type integer, real, bits, or character. It shall be an array.

DIM shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n
is the rank of ARRAY. The corresponding actual argument shall not be an optional
32 dummy argument.

33 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

1 KIND (optional) shall be a scalar integer initialization expression.

2 BACK (optional) shall be scalar and of type logical.

3 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
 4 value of KIND; otherwise the kind type parameter is that of default integer type. If DIM is absent, the
 5 result is an array of rank one and of size equal to the rank of ARRAY; otherwise, the result is of rank
 6 $n - 1$ 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.

7 **Result Value.**

8 *Case (i):* The result of MINLOC (ARRAY) is a rank-one array whose element values are the values
 9 of the subscripts of an element of ARRAY whose value equals the minimum value of all
 10 the elements of ARRAY. The i th subscript returned lies in the range 1 to e_i , where e_i is
 11 the extent of the i th dimension of ARRAY. If ARRAY has size zero, all elements of the
 12 result are zero.

13 *Case (ii):* The result of MINLOC (ARRAY, MASK = MASK) is a rank-one array whose element
 14 values are the values of the subscripts of an element of ARRAY, corresponding to a
 15 true element of MASK, whose value equals the minimum value of all such elements of
 16 ARRAY. The i th subscript returned lies in the range 1 to e_i , where e_i is the extent of
 17 the i th dimension of ARRAY. If ARRAY has size zero or every element of MASK has the
 18 value false, all elements of the result are zero.

19 *Case (iii):* If ARRAY has rank one, MINLOC (ARRAY, DIM = DIM [, MASK = MASK]) is a scalar
 20 whose value is equal to that of the first element of MINLOC (ARRAY [, MASK = MASK]).
 21 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
 22 to

$$\text{MINLOC (ARRAY (} s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n), \text{DIM}=1 \\ [, \text{MASK} = \text{MASK (} s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

25 If only one element has the minimum value, that element's subscripts are returned. Otherwise, if more
 26 than one element has the minimum value and BACK is absent or present with the value false, the
 27 element whose subscripts are returned is the first such element, taken in array element order. If BACK
 28 is present with the value true, the element whose subscripts are returned is the last such element, taken
 29 in array element order.

30 If ARRAY has type character, the result is the value that would be selected by application of intrinsic
 31 relational operators; that is, the collating sequence for characters with the kind type parameter of the
 32 arguments is applied.

33 **Examples.**

34 *Case (i):* The value of MINLOC ([4, 3, 6, 3]) is [2] and the value of MINLOC ([4, 3, 6, 3],
 35 BACK = .TRUE.) is [4].

36 *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
 37 [1, 4]. Note that this is independent of the declared lower bounds for A.

38 *Case (iii):* The value of MINLOC ([5, -9, 3], DIM = 1) is 2. If B has the value $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$,
 39 MINLOC (B, DIM = 1) is [1, 2, 1] and MINLOC (B, DIM = 2) is [3, 1]. Note that this
 40 is independent of the declared lower bounds for B.

41 **13.7.121 MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])**

1 **Description.** Minimum value of all the elements of ARRAY along dimension DIM corresponding to
2 true elements of MASK.

3 **Class.** Transformational function.

4 **Arguments.**

5 ARRAY shall be of type integer, real, bits, or character. It shall be an array.

DIM shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n
is the rank of ARRAY. The corresponding actual argument shall not be an optional
dummy argument.

7 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

8 **Result Characteristics.** The result is of the same type and type parameters as ARRAY. It is scalar
9 if DIM is absent; otherwise, the result has rank $n - 1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$
10 where $(d_1, d_2, \dots, d_n)$ is the shape of ARRAY.

11 **Result Value.**

12 *Case (i):* The result of MINVAL (ARRAY) has a value equal to the minimum value of all the
13 elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size zero and type
14 integer or real, the result has the value of the positive number of the largest magnitude
15 supported by the processor for numbers of the type and kind type parameter of ARRAY. If
16 ARRAY has size zero and type character, the result has the value of a string of characters
17 of length LEN (ARRAY), with each character equal to CHAR ($n - 1$, KIND (ARRAY)),
18 where n is the number of characters in the collating sequence for characters with the kind
19 type parameter of ARRAY. If ARRAY has size zero and type bits, the result has the
20 value NOT (BITS (B'0', KIND(ARRAY))).

21 *Case (ii):* The result of MINVAL (ARRAY, MASK = MASK) has a value equal to that of MIN-
22 VAL (PACK (ARRAY, MASK)).

23 *Case (iii):* The result of MINVAL (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to
24 that of MINVAL (ARRAY [, MASK = MASK]) if ARRAY has rank one. Otherwise, the
25 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

26 $\text{MINVAL} (\text{ARRAY} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$
27 $[\text{, MASK} = \text{MASK} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)])$.

28 If ARRAY has type character, the result is the value that would be selected by application of intrinsic
29 relational operators; that is, the collating sequence for characters with the kind type parameter of the
30 arguments is applied.

31 **Examples.**

32 *Case (i):* The value of MINVAL ([1, 2, 3]) is 1.

33 *Case (ii):* MINVAL (C, MASK = C > 0.0) is the minimum of the positive elements of C.

34 *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 =
35 2) is [1, 2].

36 13.7.122 MOD (A, P)

37 **Description.** Remainder function.

38 **Class.** Elemental function.

Arguments.

- 1
 2 A shall be of type integer or real.
 3 P shall be of the same type and kind type parameter as A. P shall not be zero.
 4 **Result Characteristics.** Same as A.
 5 **Result Value.** The value of the result is $A - \text{INT}(A/P) * P$.
 6 **Examples.** MOD (3.0, 2.0) has the value 1.0 (approximately). MOD (8, 5) has the value 3. MOD (-8, 5)
 7 has the value -3. MOD (8, -5) has the value 3. MOD (-8, -5) has the value -3.

8 13.7.123 MODULO (A, P)

- 9 **Description.** Modulo function.
 10 **Class.** Elemental function.
 11 **Arguments.**
 12 A shall be of type integer or real.
 13 P shall be of the same type and kind type parameter as A. P shall not be zero.
 14 **Result Characteristics.** Same as A.
 15 **Result Value.**
 16 *Case (i):* A is of type integer. MODULO (A, P) has the value R such that $A = Q \times P + R$, where
 17 Q is an integer, the inequalities $0 \leq R < P$ hold if $P > 0$, and $P < R \leq 0$ hold if $P < 0$.
 18 *Case (ii):* A is of type real. The value of the result is $A - \text{FLOOR}(A / P) * P$.
 19 **Examples.** MODULO (8, 5) has the value 3. MODULO (-8, 5) has the value 2. MODULO (8, -5)
 20 has the value -2. MODULO (-8, -5) has the value -3.

21 13.7.124 MOVE_ALLOC (FROM, TO)

- 22 **Description.** Moves an allocation from one allocatable object to another.
 23 **Class.** Pure subroutine.
 24 **Arguments.**
 25 FROM may be of any type and rank. It shall be allocatable. It is an INTENT(INOUT)
 argument.
 26 TO shall be type compatible (4.3.1.3) with FROM and have the same rank. It shall be
 allocatable. It shall be polymorphic if FROM is polymorphic. It is an INTENT(OUT)
 argument. Each nondeferred parameter of the declared type of TO shall have the same
 value as the corresponding parameter of the declared type of FROM.
 27 The allocation status of TO becomes unallocated if FROM is unallocated on entry to MOVE_ALLOC.
 28 Otherwise, TO becomes allocated with dynamic type, type parameters, array bounds, and value identical
 29 to those that FROM had on entry to MOVE_ALLOC.
 30 If TO has the TARGET attribute, any pointer associated with FROM on entry to MOVE_ALLOC
 31 becomes correspondingly associated with TO. If TO does not have the TARGET attribute, the pointer
 32 association status of any pointer associated with FROM on entry becomes undefined.

1 The allocation status of FROM becomes unallocated.

2 **Example.**

```

3      REAL,ALLOCATABLE :: GRID(:),TEMPGRID(:)
4      ...
5      ALLOCATE(GRID(-N:N))          ! initial allocation of GRID
6      ...
7      ! "reallocation" of GRID to allow intermediate points
8      ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
9      TEMPGRID(::2)=GRID ! distribute values to new locations
10     CALL MOVE_ALLOC(TO=GRID,FROM=TEMPGRID)
11                                     ! old grid is deallocated because TO is
12                                     ! INTENT(OUT), and GRID then "takes over"
13                                     ! new grid allocation

```

NOTE 13.17

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.

14 13.7.125 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

15 **Description.** Copies a sequence of bits from one data object to another.

16 **Class.** Elemental subroutine.

17 **Arguments.**

18	FROM	shall be of type integer or bits. It is an INTENT (IN) argument.
19	FROMPOS	shall be of type integer and nonnegative. It is an INTENT (IN) argument. FROMPOS + LEN shall be less than or equal to BIT_SIZE (FROM). The model for the interpretation of an integer value as a sequence of bits is in 13.3.
20	LEN	shall be of type integer and nonnegative. It is an INTENT (IN) argument.
21	TO	shall be a variable of the same type and kind type parameter value as FROM and may be associated with FROM (12.8.3). It is an INTENT (INOUT) argument. TO is defined by copying the sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry. The model for the interpretation of an integer value as a sequence of bits is in 13.3.
22	TOPOS	shall be of type integer and nonnegative. It is an INTENT (IN) argument. TOPOS + LEN shall be less than or equal to BIT_SIZE (TO).

23 **Examples.** If TO has the initial value 6, the value of TO after the statement
 24 CALL MVBITS (7, 2, 2, TO, 0) is 5. If TO has the initial value B'000000111111', the value of TO after
 25 the statement CALL MVBITS (B'000000000011', 0, 2, TO, 8) is B'001100111111'.

1 13.7.126 NEAREST (X, S)

2 **Description.** Returns the nearest different machine-representable number in a given direction.

3 **Class.** Elemental function.

4 **Arguments.**

5 X shall be of type real.

6 S shall be of type real and not equal to zero.

7 **Result Characteristics.** Same as X.

8 **Result Value.** The result has a value equal to the machine-representable number distinct from X and
9 nearest to it in the direction of the infinity with the same sign as S.

10 **Example.** NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$ on a machine whose representation is that of
11 the model in Note 13.3.

NOTE 13.18

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.

12 13.7.127 NEW_LINE (A)

13 **Description.** Returns a newline character.

14 **Class.** Inquiry function.

15 **Argument.** A shall be of type character. It may be a scalar or an array.

16 **Result Characteristics.** Character scalar of length one with the same kind type parameter as A.

17 **Result Value.**

18 *Case (i):* If A is of the default character type and the character in position 10 of the ASCII collating
19 sequence is representable in the default character set, then the result is ACHAR(10).

20 *Case (ii):* If A is of the ASCII character type or the ISO 10646 character type, then the result is
21 CHAR(10,KIND(A)).

22 *Case (iii):* Otherwise, the result is a processor-dependent character that represents a newline in
23 output to files connected for formatted stream output if there is such a character.

24 *Case (iv):* Otherwise, the result is the blank character.

25 13.7.128 NINT (A [, KIND])

26 **Description.** Nearest integer.

27 **Class.** Elemental function.

28 **Arguments.**

29 A shall be of type real.

30 KIND (optional) shall be a scalar integer initialization expression.

1 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
2 value of KIND; otherwise, the kind type parameter is that of default integer type.

3 **Result Value.** The result is the integer nearest A, or if there are two integers equally near A, the result
4 is whichever such integer has the greater magnitude.

5 **Example.** NINT (2.783) has the value 3.

6 **13.7.129 NOT (I)**

7 **Description.** Performs a bitwise complement.

8 **Class.** Elemental function.

9 **Argument.** I shall be of type integer or bits.

10 **Result Characteristics.** Same as I.

11 **Result Value.** The result has the value obtained by complementing I bit-by-bit according to the
12 following truth table:

I	NOT (I)
1	0
0	1

13 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

14 **Examples.** If I is represented by the string of bits 01010101, NOT (I) has the binary value 10101010.
15 NOT (Z'FFFF0000') has the value Z'0000FFFF'.

16 **13.7.130 NORM2 (X)**

17 **Description.** L_2 norm of an array.

18 **Class.** Transformational function.

19 **Argument.** X shall be of type real. It shall be an array.

20 **Result Characteristics.** Scalar of the same type and kind type parameter value as X.

21 **Result Value.** The result has a value equal to a processor-dependent approximation to the generalized
22 L_2 norm of X, which is the square root of the sum of the squares of the elements of X. It is recommended
23 that the processor compute the result without undue overflow or underflow.

24 **Example.** The value of NORM2 ([3.0, 4.0]) is 5.0 (approximately).

25 **13.7.131 NULL ([MOLD])**

26 **Description.** Returns a disassociated pointer or designates an unallocated allocatable component of a
27 structure constructor.

28 **Class.** Transformational function.

29 **Argument.** MOLD shall be a pointer or allocatable. It may be of any type or may be a procedure
30 pointer. If MOLD is a pointer its pointer association status may be undefined, disassociated, or asso-
31 ciated. If MOLD is allocatable its allocation status may be allocated or unallocated. It need not be
32 defined with a value.

- 1 **Result Characteristics.** If MOLD is present, the characteristics are the same as MOLD. If MOLD
2 has deferred type parameters, those type parameters of the result are deferred.
- 3 If MOLD is absent, the characteristics of the result are determined by the entity with which the reference
4 is associated. See Table 13.1. MOLD shall not be absent in any other context. If any type parameters
5 of the contextual entity are deferred, those type parameters of the result are deferred. If any type
6 parameters of the contextual entity are assumed, MOLD shall be present.
- 7 If the context of the reference to NULL is an actual argument to a generic procedure, MOLD shall be
8 present if the type, type parameters, or rank are required to resolve the generic reference. MOLD shall
9 also be present if the reference appears as an actual argument corresponding to a dummy argument with
10 assumed character length.

Table 13.1: **Characteristics of the result of NULL ()**

Appearance of NULL ()	Type, type parameters, and rank of result:
right side of a pointer assignment	pointer on the left side
initialization for an object in a declaration	the object
default initialization for a component	the component
in a structure constructor	the corresponding component
as an actual argument	the corresponding dummy argument
in a DATA statement	the corresponding pointer object

- 11 **Result.** The result is a disassociated pointer or an unallocated allocatable entity.

12 **Examples.**

- 13 *Case (i):* REAL, POINTER, DIMENSION(:) :: VEC => NULL () defines the initial association
14 status of VEC to be disassociated.

- 15 *Case (ii):* The MOLD argument is required in the following:

```

16      INTERFACE GEN
17          SUBROUTINE S1 (J, PI)
18              INTEGER J
19              INTEGER, POINTER :: PI
20          END SUBROUTINE S1
21          SUBROUTINE S2 (K, PR)
22              INTEGER K
23              REAL, POINTER :: PR
24          END SUBROUTINE S2
25      END INTERFACE
26      REAL, POINTER :: REAL_PTR
27      CALL GEN (7, NULL (REAL_PTR) )      ! Invokes S2

```

28 **13.7.132 NUM_IMAGES ()**

- 29 **Description.** Returns the number of images.

- 30 **Class.** Inquiry function.

- 31 **Argument.** None.

Result Characteristics. Default integer scalar.

1

2 **Result Value.** The number of images.3 **Example.** The following code uses image 1 to read data and broadcast it to other images.

```

4      REAL :: P[*]
5      IF (THIS_IMAGE()==1) THEN
6          READ (6,*) P
7          DO I = 2, NUM_IMAGES()
8              P[I] = P
9          END DO
10     END IF
11     SYNC ALL

```

12 **13.7.133 PACK (ARRAY, MASK [, VECTOR])**13 **Description.** Pack an array into an array of rank one under the control of a mask.14 **Class.** Transformational function.15 **Arguments.**16 **ARRAY** may be of any type. It shall be an array.17 **MASK** shall be of type logical and shall be conformable with ARRAY.

18 **VECTOR** shall be of the same type and type parameters as ARRAY and shall have rank one.
 (optional) VECTOR shall have at least as many elements as there are true elements in MASK.
 If MASK is scalar with the value true, VECTOR shall have at least as many elements
 as there are in ARRAY.

19 **Result Characteristics.** The result is an array of rank one with the same type and type parameters
 20 as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the
 21 number t of true elements in MASK unless MASK is scalar with the value true, in which case the result
 22 size is the size of ARRAY.

23 **Result Value.** Element i of the result is the element of ARRAY that corresponds to the i th true
 24 element of MASK, taking elements in array element order, for $i = 1, 2, \dots, t$. If VECTOR is present
 25 and has size $n > t$, element i of the result has the value VECTOR (i), for $i = t + 1, \dots, n$.

26 **Examples.** The nonzero elements of an array M with the value $\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$ may be “gathered” by the
 27 function PACK. The result of PACK (M, MASK = M /= 0) is [9, 7] and the result of PACK (M, M /= 0,
 28 VECTOR = [2, 4, 6, 8, 10, 12]) is [9, 7, 6, 8, 10, 12].

29 **13.7.134 PARITY (MASK [, DIM])**30 **Description.** Determine whether an odd number of values are true in MASK along dimension DIM.31 **Class.** Transformational function.32 **Arguments.**

MASK shall be of type logical. It shall be an array.

DIM (optional) shall be scalar and of type integer 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.

Result Characteristics. The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM is absent; 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 PARITY (MASK) has the value true if an odd number of the elements of MASK are true, and false otherwise.

Case (ii): If MASK has rank one, PARITY (MASK, DIM) is equal to PARITY (MASK). Otherwise, the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of PARITY (MASK, DIM) is equal to PARITY (MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$).

Examples.

Case (i): The value of PARITY ([T, T, T, F]) is true if T has the value true and F has the value false.

Case (ii): If B is the array $\begin{bmatrix} T & T & F \\ T & T & T \end{bmatrix}$, where T has the value true and F has the value false, then PARITY (B, DIM=1) has the value [F, F, T] and PARITY (B, DIM=2) has the value [F, T].

13.7.135 POPCNT (I)

Description. Return the number of one bits.

Class. Elemental function.

Argument. I shall be of type integer or bits.

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 the default integer type.

Result Value. If I is of type integer, the result value is equal to the number of one bits in the sequence of bits of I. The model for the interpretation of an integer value as a sequence of bits is in 13.3. If I is of type bits, the result value is the number of one bits in I.

Examples. POPCNT ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 2, 1, 2, 2]. POPCNT (Z'FFFF0000') has the value 16. If B is of type bits, POPCNT (HUGE (B)) has the same value as KIND (B).

13.7.136 POPPAR (I)

Description. Return the bitwise parity.

Class. Elemental function.

Argument. I shall be of type integer or bits.

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]. POPPAR (Z'FFFF0000') has the value 0.

1 13.7.137 PRECISION (X)

2 **Description.** Returns the decimal precision of the model representing real numbers with the same kind
3 type parameter as the argument.

4 **Class.** Inquiry function.

5 **Argument.** X shall be of type real or complex. It may be a scalar or an array.

6 **Result Characteristics.** Default integer scalar.

7 **Result Value.** The result has the value $\text{INT}((p-1) * \text{LOG}_{10}(b)) + k$, where b and p are as defined
8 in 13.4 for the model representing real numbers with the same value for the kind type parameter as X,
9 and where k is 1 if b is an integral power of 10 and 0 otherwise.

10 **Example.** PRECISION (X) has the value $\text{INT}(23 * \text{LOG}_{10}(2.)) = \text{INT}(6.92\dots) = 6$ for real X whose
11 model is as in Note 13.3.

12 13.7.138 PRESENT (A)

13 **Description.** Determine whether an optional argument is present.

14 **Class.** Inquiry function.

15 **Argument.** A shall be the name of an optional dummy argument that is accessible in the subprogram
16 in which the PRESENT function reference appears. It may be of any type and it may be a pointer. It
17 may be a scalar or an array. It may be a dummy procedure. The dummy argument A has no INTENT
18 attribute.

19 **Result Characteristics.** Default logical scalar.

20 **Result Value.** The result has the value true if A is present (12.5.2.13) and otherwise has the value
21 false.

NOTE 13.19

The PRESENT intrinsic function is also used during macro expansion (3.5.2.2).

22 13.7.139 PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])

23 **Description.** Product of all the elements of ARRAY along dimension DIM corresponding to the true
24 elements of MASK.

25 **Class.** Transformational function.

26 **Arguments.**

27 ARRAY shall be of type integer, real, or complex. It shall be an array.

DIM shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n
is the rank of ARRAY. The corresponding actual argument shall not be an optional
28 dummy argument.

29 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

30 **Result Characteristics.** The result is of the same type and kind type parameter as ARRAY. It is
31 scalar if DIM is absent; otherwise, the result has rank $n-1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$
32 $\dots, d_n)$ where $(d_1, d_2, \dots, d_n)$ is the shape of ARRAY.

1 **Result Value.**

2 *Case (i):* The result of PRODUCT (ARRAY) has a value equal to a processor-dependent approxi-
3 mation to the product of all the elements of ARRAY or has the value one if ARRAY has
4 size zero.

5 *Case (ii):* The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a processor-
6 dependent approximation to the product of the elements of ARRAY corresponding to the
7 true elements of MASK or has the value one if there are no true elements.

8 *Case (iii):* If ARRAY has rank one, PRODUCT (ARRAY, DIM = DIM [, MASK = MASK])
9 has a value equal to that of PRODUCT (ARRAY [, MASK = MASK]). Otherwise,
10 the value of element $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ of PRODUCT (ARRAY,
11 DIM = DIM [, MASK = MASK]) is equal to

12 $\text{PRODUCT (ARRAY (} s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) [, \text{MASK = MASK}$
13 $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)])$.

14 **Examples.**

15 *Case (i):* The value of PRODUCT ([1, 2, 3]) is 6.

16 *Case (ii):* PRODUCT (C, MASK = C > 0.0) forms the product of the positive elements of C.

17 *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,
18 DIM = 2) is [15, 48].

19 **13.7.140 RADIX (X)**

20 **Description.** Returns the base of the model representing numbers of the same type and kind type
21 parameter as the argument.

22 **Class.** Inquiry function.

23 **Argument.** X shall be of type integer or real. It may be a scalar or an array.

24 **Result Characteristics.** Default integer scalar.

25 **Result Value.** The result has the value r if X is of type integer and the value b if X is of type real,
26 where r and b are as defined in 13.4 for the model representing numbers of the same type and kind type
27 parameter as X.

28 **Example.** RADIX (X) has the value 2 for real X whose model is as in Note 13.3.

29 **13.7.141 RANDOM_NUMBER (HARVEST)**

30 **Description.** Returns one pseudorandom number or an array of pseudorandom numbers.

31 **Class.** Subroutine.

32 **Argument.** HARVEST shall be of type real or bits. It is an INTENT (OUT) argument. It may
33 be a scalar or an array variable. If it is real, it is assigned pseudorandom numbers from the uniform
34 distribution in the interval $0 \leq x < 1$. If it is of type bits, it is assigned pseudorandom values with each
35 of the KIND (HARVEST) bits of each value having a probability of approximately 0.5 of being 1.

36 **Examples.**

37 REAL X, Y (10, 10)

38 BITS B

```

1      ! Initialize X with a pseudorandom number
2      CALL RANDOM_NUMBER (HARVEST = X)
3      CALL RANDOM_NUMBER (Y)
4      ! X and Y contain uniformly distributed random numbers
5      CALL RANDOM_NUMBER(B)
6      ! B contains a uniformly random collection of 0 and 1 bits.

```

7 13.7.142 RANDOM_SEED ([SIZE, PUT, GET])

8 **Description.** Restarts or queries the pseudorandom number generator used by RANDOM_NUMBER.

9 **Class.** Subroutine.

10 **Arguments.** There shall either be exactly one or no arguments present.

SIZE (optional) shall be scalar and of type default integer. 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 current value of the seed.

14 If no argument is present, the processor assigns a processor-dependent value to the seed.

15 The pseudorandom number generator used by RANDOM_NUMBER maintains a seed that is updated during the execution of RANDOM_NUMBER and that may be specified or returned by RANDOM_SEED. Computation of the seed from the argument PUT is performed in a processor-dependent manner. The value returned by GET need not be the same as the value specified by PUT in an immediately preceding reference to RANDOM_SEED. For example, following execution of the statements

```

20      CALL RANDOM_SEED (PUT=SEED1)
21      CALL RANDOM_SEED (GET=SEED2)

```

22 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

```

25      CALL RANDOM_SEED (PUT=SEED1)
26      CALL RANDOM_SEED (GET=SEED2)
27      CALL RANDOM_NUMBER (X1)
28      CALL RANDOM_SEED (PUT=SEED2)
29      CALL RANDOM_NUMBER (X2)

```

30 X2 equals X1.

31 **Examples.**

```

1      CALL RANDOM_SEED                                ! Processor initialization
2      CALL RANDOM_SEED (SIZE = K)                      ! Puts size of seed in K
3      CALL RANDOM_SEED (PUT = SEED (1 : K)) ! Define seed
4      CALL RANDOM_SEED (GET = OLD (1 : K)) ! Read current seed

```

5 13.7.143 RANGE (X)

6 **Description.** Returns the decimal exponent range of the model representing integer or real numbers
7 with the same kind type parameter as the argument.

8 **Class.** Inquiry function.

9 **Argument.** X shall be of type integer, real, or complex. It may be a scalar or an array.

10 **Result Characteristics.** Default integer scalar.

11 **Result Value.**

- 12 *Case (i):* For an integer argument, the result has the value INT (LOG10 (HUGE(X))).
 13 *Case (ii):* For a real argument, the result has the value INT (MIN (LOG10 (HUGE(X)), -LOG10
 14 (TINY(X)))).
 15 *Case (iii):* For a complex argument, the result has the value RANGE(REAL(X)).

16 **Examples.** RANGE (X) has the value 38 for real X whose model is as in Note 13.3, because in this
 17 case $\text{HUGE}(X) = (1 - 2^{-24}) \times 2^{127}$ and $\text{TINY}(X) = 2^{-127}$.

18 13.7.144 REAL (A [, KIND])

19 **Description.** Convert to real type.

20 **Class.** Elemental function.

21 **Arguments.**

22 A shall be of type integer, real, complex, or bits.

23 KIND (optional) shall be a scalar integer initialization expression.

24 **Result Characteristics.** Real.

- 25 *Case (i):* If A is of type integer, real, or bits, and KIND is present, the kind type parameter is that
 26 specified by the value of KIND. If A is of type integer, real, or bits, and KIND is not
 27 present, the kind type parameter is that of default real type.
 28 *Case (ii):* If A is of type complex and KIND is present, the kind type parameter is that specified
 29 by the value of KIND. If A is of type complex and KIND is not present, the kind type
 30 parameter is the kind type parameter of A.

31 **Result Value.**

- 32 *Case (i):* If A is of type integer or real, the result is equal to a processor-dependent approximation
 33 to A.
 34 *Case (ii):* If A is of type complex, the result is equal to a processor-dependent approximation to the
 35 real part of A.
 36 *Case (iii):* If A is of type bits and KIND (A) is greater than or equal to BITS_KIND (*result*), the
 37 result has the value whose physical representation is the same as the rightmost bits of A.

1 *Case (iv):* If A is of type bits and KIND (A) is less than BITS_KIND (*result*), the rightmost
 2 KIND (A) bits of the physical representation of the result value are the same as those of
 3 A, and the remaining bits of the physical representation of the result value are zero.

4 **Examples.** REAL (−3) has the value −3.0. REAL (Z) has the same kind type parameter and the same
 5 value as the real part of the complex variable Z. REAL (Z'7F800000') has the value positive infinity if
 6 the default real kind is IEEE single precision.

7 13.7.145 REPEAT (STRING, NCOPIES)

8 **Description.** Concatenate several copies of a string.

9 **Class.** Transformational function.

10 **Arguments.**

11 STRING shall be scalar and of type character.

12 NCOPIES shall be scalar and of type integer. Its value shall not be negative.

13 **Result Characteristics.** Character scalar of length NCOPIES times that of STRING, with the same
 14 kind type parameter as STRING.

15 **Result Value.** The value of the result is the concatenation of NCOPIES copies of STRING.

16 **Examples.** REPEAT ('H', 2) has the value HH. REPEAT ('XYZ', 0) has the value of a zero-length
 17 string.

18 13.7.146 RESHAPE (SOURCE, SHAPE [, PAD, ORDER])

19 **Description.** Constructs an array of a specified shape from the elements of a given array.

20 **Class.** Transformational function.

21 **Arguments.**

22 SOURCE may be of any type. It shall be an array. 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.

23 SHAPE shall be of type integer, rank one, and constant size. Its size shall be positive and less
 than 8. It shall not have an element whose value is negative.

24 PAD (optional) shall be of the same type and type parameters as SOURCE. It shall be an array.

25 ORDER shall be of type integer, shall have the same shape as SHAPE, and its value shall be
 (optional) 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*).

26 **Result Characteristics.** The result is an array of shape SHAPE (that is, SHAPE (RESHAPE (SOUR-
 27 CE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

28 **Result Value.** The elements of the result, taken in permuted subscript order ORDER (1), ..., OR-
 29 DER (*n*), are those of SOURCE in normal array element order followed if necessary by those of PAD in
 30 array element order, followed if necessary by additional copies of PAD in array element order.

31 **Examples.** RESHAPE ([1, 2, 3, 4, 5, 6], [2, 3]) has the value $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$.

1 RESHAPE ([1, 2, 3, 4, 5, 6], [2, 4], [0, 0], [2, 1]) has the value $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$.

2 13.7.147 RRSPACING (X)

3 **Description.** Returns the reciprocal of the relative spacing of model numbers near the argument value.

4 **Class.** Elemental function.

5 **Argument.** X shall be of type real.

6 **Result Characteristics.** Same as X.

7 **Result Value.** The result has the value $|X \times b^{-e}| \times b^p$, where b , e , and p are as defined in 13.4 for the
 8 value nearest to X in the model for real values whose kind type parameter is that of X; if there are two
 9 such values, the value of greater absolute value is taken. If X is an IEEE infinity, the result is zero. If
 10 X is an IEEE NaN, the result is that NaN.

J3 internal note

Unresolved Technical Issue 092

This is contradictory for IEEE infinity: the specific sentence contradicts the formula, which would imply that $\text{RRSPACING}(\pm\infty)=\text{NaN}$.

Moreover, it appears to me that there is no limit as $X \rightarrow \infty$ for RRSPACING , and so the value implied by the formula (NaN) is indeed the correct value for $\text{RRSPACING}(\pm\infty)$.

11 **Example.** $\text{RRSPACING}(-3.0)$ has the value 0.75×2^{24} for reals whose model is as in Note 13.3.

12 13.7.148 SAME_TYPE_AS (A, B)

13 **Description.** Inquires whether the dynamic type of A is the same as the dynamic type of B.

14 **Class.** Inquiry function.

15 **Arguments.**

16 A shall be an object of extensible type. If it is a pointer, it shall not have an undefined
 association status.

17 B shall be an object of extensible type. If it is a pointer, it shall not have an undefined
 association status.

18 **Result Characteristics.** Default logical scalar.

19 **Result Value.** The result is true if and only if the dynamic type of A is the same as the dynamic type
 20 of B.

NOTE 13.20

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

21 13.7.149 SCALE (X, I)

22 **Description.** Returns $X \times b^I$ where b is the base of the model representation of X.

23 **Class.** Elemental function.

24 **Arguments.**

25 X shall be of type real.

1 I shall be of type integer.

2 **Result Characteristics.** Same as X.

3 **Result Value.** The result has the value $X \times b^I$, where b is defined in 13.4 for model numbers representing
4 values of X, provided this result is within range; if not, the result is processor dependent.

5 **Example.** SCALE (3.0, 2) has the value 12.0 for reals whose model is as in Note 13.3.

6 **13.7.150 SCAN (STRING, SET [, BACK, KIND])**

7 **Description.** Scan a string for any one of the characters in a set of characters.

8 **Class.** Elemental function.

9 **Arguments.**

10 STRING shall be of type character.

11 SET shall be of type character with the same kind type parameter as STRING.

12 BACK (optional) shall be of type logical.

13 KIND (optional) shall be a scalar integer initialization expression.

14 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
15 value of KIND; otherwise the kind type parameter is that of default integer type.

16 **Result Value.**

17 *Case (i):* If BACK is absent or is present with the value false and if STRING contains at least one
18 character that is in SET, the value of the result is the position of the leftmost character
19 of STRING that is in SET.

20 *Case (ii):* If BACK is present with the value true and if STRING contains at least one character that
21 is in SET, the value of the result is the position of the rightmost character of STRING
22 that is in SET.

23 *Case (iii):* The value of the result is zero if no character of STRING is in SET or if the length of
24 STRING or SET is zero.

25 **Examples.**

26 *Case (i):* SCAN ('FORTRAN', 'TR') has the value 3.

27 *Case (ii):* SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

28 *Case (iii):* SCAN ('FORTRAN', 'BCD') has the value 0.

29 **13.7.151 SELECTED_BITS_KIND (N)**

30 **Description.** Returns a value of the kind type parameter of a bits type with N bits.

31 **Class.** Transformational function.

32 **Argument.** N shall be scalar and of type integer.

33 **Result Characteristics.** Default integer scalar.

34 **Result Value.** The result value is N if the processor supports a bits type with a kind type parameter
35 equal to N; otherwise the result value is -1 .

1 **Example.** If the value of NUMERIC_STORAGE_SIZE is 32, SELECTED_BITS_KIND (43) has the
2 value 43.

3 13.7.152 SELECTED_CHAR_KIND (NAME)

4 **Description.** Returns the value of the kind type parameter of the character set named by the argument.

5 **Class.** Transformational function.

6 **Argument.** NAME shall be scalar and of type default character.

7 **Result Characteristics.** Default integer scalar.

8 **Result Value.** If NAME has the value DEFAULT, then the result has a value equal to that of the kind
9 type parameter of the default character type. If NAME has the value ASCII, then the result has a value
10 equal to that of the kind type parameter of the ASCII character type if the processor supports such a
11 type; otherwise the result has the value -1. If NAME has the value ISO_10646, then the result has a
12 value equal to that of the kind type parameter of the ISO/IEC 10646-1:2000 UCS-4 character type if the
13 processor supports such a type; otherwise the result has the value -1. If NAME is a processor-defined
14 name of some other character type supported by the processor, then the result has a value equal to that
15 of the kind type parameter of that character type. If NAME is not the name of a supported character
16 type, then the result has the value -1. The NAME is interpreted without respect to case or trailing
17 blanks.

18 **Examples.** SELECTED_CHAR_KIND ('ASCII') has the value 1 on a processor that uses 1 as the kind
19 type parameter for the ASCII character set. The following subroutine produces a Japanese date stamp.

```
20      SUBROUTINE create_date_string(string)
21          INTRINSIC date_and_time,selected_char_kind
22          INTEGER,PARAMETER :: ucs4 = selected_char_kind("ISO_10646")
23          CHARACTER(1,UCS4),PARAMETER :: nen=CHAR(INT(Z'5e74'),UCS4), & !year
24             gatsu=CHAR(INT(Z'6708'),UCS4), & !month
25             nichu=CHAR(INT(Z'65e5'),UCS4) !day
26          CHARACTER(len= *, kind= ucs4) string
27          INTEGER values(8)
28          CALL date_and_time(values=values)
29          WRITE(string,1) values(1),nen,values(2),gatsu,values(3),nichu
30      1 FORMAT(IO,A,IO,A,IO,A)
31      END SUBROUTINE"
```

32 13.7.153 SELECTED_INT_KIND (R)

33 **Description.** Returns a value of the kind type parameter of an integer type that represents all integer
34 values n with $-10^R < n < 10^R$.

35 **Class.** Transformational function.

36 **Argument.** R shall be scalar and of type integer.

37 **Result Characteristics.** Default integer scalar.

38 **Result Value.** The result has a value equal to the value of the kind type parameter of an integer type
39 that represents all values n in the range $-10^R < n < 10^R$, or if no such kind type parameter is available

1 on the processor, the result is -1 . If more than one kind type parameter meets the criterion, the value
 2 returned is the one with the smallest decimal exponent range, unless there are several such values, in
 3 which case the smallest of these kind values is returned.

4 **Example.** Assume a processor supports two integer kinds, 32 with representation method $r = 2$ and
 5 $q = 31$, and 64 with representation method $r = 2$ and $q = 63$. On this processor `SELECTED_INT_-`
 6 `KIND(9)` has the value 32 and `SELECTED_INT_KIND(10)` has the value 64.

7 **13.7.154 SELECTED_REAL_KIND ([P, R, RADIX])**

8 **Description.** Returns a value of the kind type parameter of a real type with decimal precision of at
 9 least P digits, a decimal exponent range of at least R, and a radix of RADIX.

10 **Class.** Transformational function.

11 **Arguments.** At least one argument shall be present.

12 P (optional) shall be scalar and of type integer.

13 R (optional) shall be scalar and of type integer.

14 RADIX (optional) shall be scalar and of type integer.

15 **Result Characteristics.** Default integer scalar.

16 **Result Value.** If P or R is absent, the result value is the same as if it were present with the value zero.
 17 If RADIX is absent, there is no requirement on the radix of the selected kind.

18 The result has a value equal to a value of the kind type parameter of a real type with decimal precision,
 19 as returned by the function `PRECISION`, of at least P digits, a decimal exponent range, as returned by
 20 the function `RANGE`, of at least R, and a radix, as returned by the function `RADIX`, of RADIX, if such
 21 a kind type parameter is available on the processor.

22 Otherwise, the result is -1 if the processor supports a real type with radix RADIX and exponent range
 23 of at least R but not with precision of at least P, -2 if the processor supports a real type with radix
 24 RADIX and precision of at least P but not with exponent range of at least R, -3 if the processor
 25 supports a real type with radix RADIX but with neither precision of at least P nor exponent range of
 26 at least R, -4 if the processor supports a real type with radix RADIX and either precision of at least
 27 P or exponent range of at least R but not both together, and -5 if the processor supports no real type
 28 with radix RADIX.

29 If more than one kind type parameter value meets the criteria, the value returned is the one with the
 30 smallest decimal precision, unless there are several such values, in which case the smallest of these kind
 31 values is returned.

32 **Example.** `SELECTED_REAL_KIND (6, 70)` has the value `KIND (0.0)` on a machine that supports a
 33 default real approximation method with $b = 16$, $p = 6$, $e_{\min} = -64$, and $e_{\max} = 63$ and does not have
 34 a less precise approximation method.

35 **13.7.155 SET_EXPONENT (X, I)**

36 **Description.** Returns the model number whose fractional part is the fractional part of the model
 37 representation of X and whose exponent part is I.

38 **Class.** Elemental function.

39 **Arguments.**

1 X shall be of type real.

2 I shall be of type integer.

3 **Result Characteristics.** Same as X.

4 **Result Value.** The result has the value $X \times b^{I-e}$, where b and e are as defined in 13.4 for the representation for the value of X in the model that has the radix of X but no limits on exponent values. If X
5 has the value zero, the result has the same value as X.
6

J3 internal note

Unresolved Technical Issue 093

The above is broken for Inf and NaN. I suggest SET_EXPONENT(Inf or NaN,any) = NaN.

7 **Example.** SET_EXPONENT (3.0, 1) has the value 1.5 for reals whose model is as in Note 13.3.

8 13.7.156 SHAPE (SOURCE [, KIND])

9 **Description.** Returns the shape of an array or a scalar.

10 **Class.** Inquiry function.

11 **Arguments.**

12 SOURCE may be of any type. It may be a scalar or an array. It shall not be an unallocated
allocatable or a pointer that is not associated. It shall not be an assumed-size array.

13 KIND (optional) shall be a scalar integer initialization expression.

14 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
15 value of KIND; otherwise the kind type parameter is that of default integer type. The result is an array
16 of rank one whose size is equal to the rank of SOURCE.

17 **Result Value.** The value of the result is the shape of SOURCE.

18 **Examples.** The value of SHAPE (A (2:5, -1:1)) is [4, 3]. The value of SHAPE (3) is the rank-one
19 array of size zero.

20 13.7.157 SHIFTA (I, SHIFT)

21 **Description.** Performs a right shift with fill.

22 **Class.** Elemental function.

23 **Arguments.**

24 I shall be of type integer or bits.

25 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

26 **Result Characteristics.** Same as I.

27 **Result Value.** The result has the value obtained by shifting the bits of I to the right SHIFT bits and
28 replicating the leftmost bit of I in the left SHIFT bits.

29 If SHIFT is zero the result is I. Bits shifted out from the right are lost. The model for the interpretation
30 of an integer value as a sequence of bits is in 13.3.

31 **Example.** SHIFTA (B'10000', 2) has the value B'11100'.

1 13.7.158 SHIFTL (I, SHIFT)

2 **Description.** Performs a left shift.

3 **Class.** Elemental function.

4 **Arguments.**

5 I shall be of type integer or bits.

6 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

7 **Result Characteristics.** Same as I.

8 **Result Value.** The result has a value equal to ISHFT (I, SHIFT).

9 **Examples.** SHIFTL (3, 1) has the value 6. SHIFTL (B'00001', 2) has the value B'00100'.

10 13.7.159 SHIFTR (I, SHIFT)

11 **Description.** Performs a right shift.

12 **Class.** Elemental function.

13 **Arguments.**

14 I shall be of type integer or bits.

15 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT_SIZE (I).

16 **Result Characteristics.** Same as I.

17 **Result Value.** The value of the result is ISHFT (I, -SHIFT).

18 **Examples.** SHIFTR (3, 1) has the value 1. SHIFTR (B'10000', 2) has the value B'00100'.

19 13.7.160 SIGN (A, B)

20 **Description.** Magnitude of A with the sign of B.

21 **Class.** Elemental function.

22 **Arguments.**

23 A shall be of type integer or real.

24 B shall be of the same type and kind type parameter as A.

25 **Result Characteristics.** Same as A.

26 **Result Value.**

27 *Case (i):* If $B > 0$, the value of the result is $|A|$.

28 *Case (ii):* If $B < 0$, the value of the result is $-|A|$.

29 *Case (iii):* If B is of type integer and $B=0$, the value of the result is $|A|$.

30 *Case (iv):* If B is of type real and is zero, then

31 (1) If the processor cannot distinguish between positive and negative real zero, the
32 value of the result is $|A|$.

33 (2) If B is positive real zero, the value of the result is $|A|$.

1 (3) If B is negative real zero, the value of the result is $-|A|$.

2 **Example.** SIGN (-3.0, 2.0) has the value 3.0.

3 **13.7.161 SIN (X)**

4 **Description.** Sine function.

5 **Class.** Elemental function.

6 **Argument.** X shall be of type real or complex.

7 **Result Characteristics.** Same as X.

8 **Result Value.** The result has a value equal to a processor-dependent approximation to $\sin(X)$. If X is
9 of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a
10 value in radians.

11 **Example.** SIN (1.0) has the value 0.84147098 (approximately).

12 **13.7.162 SINH (X)**

13 **Description.** Hyperbolic sine function.

14 **Class.** Elemental function.

15 **Argument.** X shall be of type real or complex.

16 **Result Characteristics.** Same as X.

17 **Result Value.** The result has a value equal to a processor-dependent approximation to $\sinh(X)$. If X
18 is of type complex its imaginary part is regarded as a value in radians.

19 **Example.** SINH (1.0) has the value 1.1752012 (approximately).

20 **13.7.163 SIZE (ARRAY [, DIM, KIND])**

21 **Description.** Returns the extent of an array along a specified dimension or the total number of elements
22 in the array.

23 **Class.** Inquiry function.

24 **Arguments.**

25 ARRAY may be of any type. It shall be an array. It shall not be an unallocated allocatable
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.

26 DIM (optional) shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is
the rank of ARRAY.

27 KIND (optional) shall be a scalar integer initialization expression.

28 **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that specified
29 by the value of KIND; otherwise the kind type parameter is that of default integer type.

30 **Result Value.** The result has a value equal to the extent of dimension DIM of ARRAY or, if DIM is
31 absent, the total number of elements of ARRAY.

1 **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.

2 13.7.164 SPACING (X)

3 **Description.** Returns the absolute spacing of model numbers near the argument value.

4 **Class.** Elemental function.

5 **Argument.** X shall be of type real.

6 **Result Characteristics.** Same as X.

7 **Result Value.** If X does not have the value zero and is not an IEEE infinity or NaN, the result has the
8 value $b^{\max(e-p, e_{\text{MIN}}-1)}$, where b , e , and p are as defined in 13.4 for the value nearest to X in the model
9 for real values whose kind type parameter is that of X; if there are two such values the value of greater
10 absolute value is taken. If X has the value zero, the result is the same as that of TINY (X). If X is an
11 IEEE infinity, the result is positive infinity. If X is an IEEE NaN, the result is that NaN.

12 **Example.** SPACING (3.0) has the value 2^{-22} for reals whose model is as in Note 13.3.

13 13.7.165 SPREAD (SOURCE, DIM, NCOPIES)

14 **Description.** Replicates an array by adding a dimension. Broadcasts several copies of SOURCE along
15 a specified dimension (as in forming a book from copies of a single page) and thus forms an array of
16 rank one greater.

17 **Class.** Transformational function.

18 **Arguments.**

19 SOURCE may be of any type. It may be a scalar or an array. The rank of SOURCE shall be
less than 15.

20 DIM shall be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n + 1$, where n
is the rank of SOURCE.

21 NCOPIES shall be scalar and of type integer.

22 **Result Characteristics.** The result is an array of the same type and type parameters as SOURCE
23 and of rank $n + 1$, where n is the rank of SOURCE.

24 *Case (i):* If SOURCE is scalar, the shape of the result is (MAX (NCOPIES, 0)).

25 *Case (ii):* If SOURCE is an array with shape $(d_1, d_2, \dots, d_n)$, the shape of the result is $(d_1, d_2,$
26 $\dots, d_{\text{DIM}-1}, \text{MAX (NCOPIES, 0)}, d_{\text{DIM}}, \dots, d_n)$.

27 **Result Value.**

28 *Case (i):* If SOURCE is scalar, each element of the result has a value equal to SOURCE.

29 *Case (ii):* If SOURCE is an array, the element of the result with subscripts $(r_1, r_2, \dots, r_{n+1})$ has
30 the value SOURCE $(r_1, r_2, \dots, r_{\text{DIM}-1}, r_{\text{DIM}+1}, \dots, r_{n+1})$.

31 **Examples.** If A is the array [2, 3, 4], SPREAD (A, DIM=1, NCOPIES=NC) is the array $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$
32 if NC has the value 3 and is a zero-sized array if NC has the value 0.

33 13.7.166 SQRT (X)

34 **Description.** Square root.

1 **Class.** Elemental function.

2 **Argument.** X shall be of type real or complex. Unless X is complex, its value shall be greater than or
3 equal to zero.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to the square root
6 of X. A result of type complex is the principal value with the real part greater than or equal to zero.
7 When the real part of the result is zero, the imaginary part has the same sign as the imaginary part of
8 X.

9 **Example.** SQRT (4.0) has the value 2.0 (approximately).

10 **13.7.167 STORAGE_SIZE (A [, KIND])**

11 **Description.** Returns the storage size in bits that an array element of the same dynamic type and type
12 parameters of A would have.

13 **Class.** Inquiry function.

14 **Arguments.**

15 A may be of any type. It may be a scalar or an array. If it is polymorphic it shall not
be an undefined pointer. If it has any deferred type parameters it shall not be an
unallocated allocatable or a disassociated or undefined pointer.

16 KIND (optional) shall be a scalar integer initialization expression.

17 **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that specified
18 by the value of KIND; otherwise, the kind type parameter is that of default integer type.

19 **Result Value.** The result value is the size expressed in bits for an element of an array that has
20 the dynamic type and type parameters of A. If the type and type parameters are such that storage
21 association (16.5.3) applies, the result is consistent with the named constants defined in the intrinsic
22 module ISO_FORTRAN_ENV.

NOTE 13.21

An array element might take more bits to store than an isolated scalar, since any hardware-imposed alignment requirements for array elements might not apply to a simple scalar variable.

NOTE 13.22

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.

23 **Example.** STORAGE_SIZE(1.0) has the same value as the named constant NUMERIC_STORAGE_-
24 SIZE in the intrinsic module ISO_FORTRAN_ENV.

25 **13.7.168 SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])**

26 **Description.** Sum all the elements of ARRAY along dimension DIM corresponding to the true elements
27 of MASK.

28 **Class.** Transformational function.

1 **Arguments.**2 **ARRAY** shall be of type integer, real, or complex. It shall be an array.3 **DIM** shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of **ARRAY**. The corresponding actual argument shall not be an optional dummy argument.4 **MASK** (optional) shall be of type logical and shall be conformable with **ARRAY**.5 **Result Characteristics.** The result is of the same type and kind type parameter as **ARRAY**. It is
6 scalar if **DIM** is absent; otherwise, the result has rank $n - 1$ and shape $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$
7 $\dots, d_n)$ where $(d_1, d_2, \dots, d_n)$ is the shape of **ARRAY**.8 **Result Value.**9 *Case (i):* The result of **SUM** (**ARRAY**) has a value equal to a processor-dependent approximation
10 to the sum of all the elements of **ARRAY** or has the value zero if **ARRAY** has size zero.11 *Case (ii):* The result of **SUM** (**ARRAY**, **MASK** = **MASK**) has a value equal to a processor-dependent
12 approximation to the sum of the elements of **ARRAY** corresponding to the true elements
13 of **MASK** or has the value zero if there are no true elements.14 *Case (iii):* If **ARRAY** has rank one, **SUM** (**ARRAY**, **DIM** = **DIM** [, **MASK** = **MASK**]) has a value
15 equal to that of **SUM** (**ARRAY** [, **MASK** = **MASK**]). Otherwise, the value of element
16 $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ **SUM** (**ARRAY**, **DIM** = **DIM** [, **MASK** = **MASK**])
17 is equal to18 $\text{SUM} (\text{ARRAY} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) [, \text{MASK} = \text{MASK} (s_1, s_2,$
19 $\dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)])$.20 **Examples.**21 *Case (i):* The value of **SUM** ([1, 2, 3]) is 6.22 *Case (ii):* **SUM** (**C**, **MASK** = **C** > 0.0) forms the sum of the positive elements of **C**.23 *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
24 [9, 12].25 **13.7.169 SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])**26 **Description.** Returns numeric data from a real-time clock.27 **Class.** Subroutine.28 **Arguments.**29 **COUNT** (optional) shall be scalar and of type integer. It is an **INTENT (OUT)** argument. It is assigned a processor-dependent value based on the current value of the processor clock, or – **HUGE (COUNT)** if there is no clock. The processor-dependent value is incremented by one for each clock count until the value **COUNT_MAX** is reached and is reset to zero at the next count. It lies in the range 0 to **COUNT_MAX** if there is a clock.30 **COUNT_RATE** (optional) shall be scalar and of type integer or real. 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.31 **COUNT_MAX** (optional) shall be scalar and of type integer. It is an **INTENT(OUT)** argument. It is assigned the maximum value that **COUNT** can have, or zero if there is no clock.

1 **Example.** If the processor clock is a 24-hour clock that registers time at approximately 18.20648193
2 ticks per second, at 11:30 A.M. the reference

3 CALL SYSTEM_CLOCK (COUNT = C, COUNT_RATE = R, COUNT_MAX = M)

4 defines $C = (11 \times 3600 + 30 \times 60) \times 18.20648193 = 753748$, $R = 18.20648193$, and $M = 24 \times 3600 \times$
5 $18.20648193 - 1 = 1573039$.

6 **13.7.170 TAN (X)**

7 **Description.** Tangent function.

8 **Class.** Elemental function.

9 **Argument.** X shall be of type real or complex.

10 **Result Characteristics.** Same as X.

11 **Result Value.** The result has a value equal to a processor-dependent approximation to $\tan(X)$. If X is
12 of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a
13 value in radians.

14 **Example.** TAN (1.0) has the value 1.5574077 (approximately).

15 **13.7.171 TANH (X)**

16 **Description.** Hyperbolic tangent function.

17 **Class.** Elemental function.

18 **Argument.** X shall be of type real or complex.

19 **Result Characteristics.** Same as X.

20 **Result Value.** The result has a value equal to a processor-dependent approximation to $\tanh(X)$. If X
21 is of type complex its imaginary part is regarded as a value in radians.

22 **Example.** TANH (1.0) has the value 0.76159416 (approximately).

23 **13.7.172 TEAM_IMAGES (TEAM)**

24 **Description.** Returns a list of the images in a team.

25 **Class.** Transformational function.

26 **Argument.** TEAM shall be a scalar of type IMAGE_TEAM (13.8.2.7).

27 **Result Characteristics.** The result is a default integer array of rank one and of size equal to the
28 number of images in the team identified by TEAM.

29 **Result Value.** The result is a rank-one array whose element values are the indices, in increasing order,
30 of the images in the team identified by TEAM.

31 **Examples.** If the value of TEAM was defined by the statement CALL FORM_TEAM (TEAM, [4, 2, 1])
32 then TEAM_IMAGES (TEAM) has the value [1, 2, 4]. For a value of TEAM that identifies an empty
33 image set, the result is an array of size zero.

34 **13.7.173 THIS_IMAGE () or THIS_IMAGE (CO_ARRAY [, DIM])**

1 **Description.** Returns the index of the invoking image, a single co-subscript, or a list of co-subscripts.

2 **Class.** Inquiry function.

3 **Arguments.**

4 CO_ARRAY shall be a co-array and may be of any type. If it is allocatable it shall be allocated.

DIM (optional) shall be scalar and of type default integer. Its value shall be in the range $1 \leq \text{DIM} \leq n$, where n is the co-rank of CO_ARRAY. It shall not be an absent optional dummy argument.

6 **Result Characteristics.** Type default integer. It is scalar if CO_ARRAY is absent or DIM is present; otherwise, the result has rank one and its size is equal to the co-rank of CO_ARRAY.

8 **Result Value.**

9 *Case (i):* The result of THIS_IMAGE () is a scalar with a value equal to the index of the invoking image.

11 *Case (ii):* The result of THIS_IMAGE (CO_ARRAY) is the sequence of co-subscript values for CO_ARRAY that would specify the invoking image.

13 *Case (iii):* The result of THIS_IMAGE (CO_ARRAY, DIM) is the value of co-subscript DIM in the sequence of co-subscript values for CO_ARRAY that would specify the invoking image.

15 **Examples.** If A is declared by the statement

16 REAL A (10, 20) [10, 0:9, 0:*]

17 then on image 5, THIS_IMAGE () has the value 5 and THIS_IMAGE (A) has the value [5, 0, 0]. For the same co-array on image 213, THIS_IMAGE (A) has the value [3, 1, 2].

19 The following code uses image 1 to read data. The other images then copy the data.

```
20      IF (THIS_IMAGE()==1) READ (*,*) P
21      SYNC ALL
22      P = P[1]
```

NOTE 13.23

For an example of a module that implements a function similar to the intrinsic THIS_IMAGE, see subclause C.10.1.

23 13.7.174 TINY (X)

24 **Description.** Returns the smallest positive number of the model representing numbers of the same type and kind type parameter as the argument.

26 **Class.** Inquiry function.

27 **Argument.** X shall be of type real. It may be a scalar or an array.

28 **Result Characteristics.** Scalar with the same type and kind type parameter as X.

29 **Result Value.** The result has the value $b^{e_{\min}-1}$ where b and $e_{\min}$ are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

Example. TINY (X) has the value 2^{-127} for real X whose model is as in Note 13.3.

13.7.175 TRAILZ (I)

Description. Count the number of trailing zero bits.

Class. Elemental function.

Argument. I shall be of type integer or bits.

Result Characteristics. Default integer.

Result Value. If all of the bits of I are zero, the result value is BIT.SIZE (I). Otherwise, the result value is the position of the rightmost 1 bit in I. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

Examples. TRAILZ (8) has the value 3. TRAILZ (B'101101000') has the value 3.

13.7.176 TRANSFER (SOURCE, MOLD [, SIZE])

Description. Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

Class. Transformational function.

Arguments.

SOURCE may be of any type. It may be a scalar or an array.

MOLD may be of any type. It may be a scalar or an array. If it is a variable, it need not be defined.

SIZE (optional) shall be scalar and of type integer. The corresponding actual argument shall not be an optional dummy argument.

Result Characteristics. The result is of the same type and type parameters as MOLD.

Case (i): If MOLD is a scalar and SIZE is absent, the result is a scalar.

Case (ii): If MOLD is an array and SIZE is absent, the result is an array and of rank one. Its size is as small as possible such that its physical representation is not shorter than that of SOURCE.

Case (iii): If SIZE is present, the result is an array of rank one and size SIZE.

Result Value. If the physical representation of the result has the same length as that of SOURCE, the physical representation of the result is that of SOURCE. If the physical representation of the result is longer than that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is processor dependent. If the physical representation of the result is shorter than that of SOURCE, the physical representation of the result is the leading part of SOURCE. If D and E are scalar variables such that the physical representation of D is as long as or longer than that of E, the value of TRANSFER (TRANSFER (E, D), E) shall be the value of E. IF D is an array and E is an array of rank one, the value of TRANSFER (TRANSFER (E, D), E, SIZE (E)) shall be the value of E.

Examples.

Case (i): TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000.

Case (ii): TRANSFER ([1.1, 2.2, 3.3], [(0.0, 0.0)]) is a complex rank-one array of length two whose first element has the value (1.1, 2.2) and whose second element has a real part with the

1 value 3.3. The imaginary part of the second element is processor dependent.
 2 *Case (iii):* TRANSFER ([1.1, 2.2, 3.3], [(0.0, 0.0)], 1) is a complex rank-one array of length one whose
 3 only element has the value (1.1, 2.2).

4 **13.7.177 TRANSPOSE (MATRIX)**

5 **Description.** Transpose an array of rank two.

6 **Class.** Transformational function.

7 **Argument.** MATRIX may be of any type and shall have rank two.

8 **Result Characteristics.** The result is an array of the same type and type parameters as MATRIX
 9 and with rank two and shape (n, m) where (m, n) is the shape of MATRIX.

10 **Result Value.** Element (i, j) of the result has the value $\text{MATRIX}(j + \text{LBOUND}(\text{MATRIX}, 1) - 1,$
 11 $i + \text{LBOUND}(\text{MATRIX}, 2) - 1)$.

12 **Example.** If A is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then TRANSPOSE (A) has the value $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

13 **13.7.178 TRIM (STRING)**

14 **Description.** Returns the argument with trailing blank characters removed.

15 **Class.** Transformational function.

16 **Argument.** STRING shall be of type character and shall be scalar.

17 **Result Characteristics.** Character with the same kind type parameter value as STRING and with a
 18 length that is the length of STRING less the number of trailing blanks in STRING.

19 **Result Value.** The value of the result is the same as STRING except any trailing blanks are removed.
 20 If STRING contains no nonblank characters, the result has zero length.

21 **Example.** TRIM (' A B ') has the value ' A B'.

22 **13.7.179 UBOUND (ARRAY [, DIM, KIND])**

23 **Description.** Returns all the upper bounds of an array or a specified upper bound.

24 **Class.** Inquiry function.

25 **Arguments.**

26 **ARRAY** may be of any type. It shall be an array. It shall not be an unallocated allocatable
 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.

27 **DIM (optional)** shall be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n
 is the rank of ARRAY. The corresponding actual argument shall not be an optional
 dummy argument.

28 **KIND (optional)** shall be a scalar integer initialization expression.

29 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the
 30 value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if

1 DIM is present; otherwise, the result is an array of rank one and size n , where n is the rank of ARRAY.

2 **Result Value.**

3 *Case (i):* For an array section or for an array expression, other than a whole array or array structure
 4 component, UBOUND (ARRAY, DIM) has a value equal to the number of elements in
 5 the given dimension; otherwise, it has a value equal to the upper bound for subscript DIM
 6 of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero
 7 if dimension DIM has size zero.

8 *Case (ii):* UBOUND (ARRAY) has a value whose i th element is equal to UBOUND (ARRAY, i),
 9 for $i = 1, 2, \dots, n$, where n is the rank of ARRAY.

10 **Examples.** If A is declared by the statement

11 REAL A (2:3, 7:10)

12 then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

13 **13.7.180 UNPACK (VECTOR, MASK, FIELD)**

14 **Description.** Unpack an array of rank one into an array under the control of a mask.

15 **Class.** Transformational function.

16 **Arguments.**

17 VECTOR may be of any type. It shall have rank one. Its size shall be at least t where t is the
 18 number of true elements in MASK.

18 MASK shall be an array of type logical.

19 FIELD shall be of the same type and type parameters as VECTOR and shall be conformable
 20 with MASK.

21 **Result Characteristics.** The result is an array of the same type and type parameters as VECTOR
 22 and the same shape as MASK.

23 **Result Value.** The element of the result that corresponds to the i th true element of MASK, in array
 24 element order, has the value VECTOR (i) for $i = 1, 2, \dots, t$, where t is the number of true values
 25 in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to the corresponding
 26 element of FIELD if it is an array.

26 **Examples.** Particular values may be “scattered” to particular positions in an array by using UNPACK.

27 If M is the array $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, V is the array [1, 2, 3], and Q is the logical mask $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$, where

28 “T” represents true and “.” represents false, then the result of UNPACK (V, MASK = Q, FIELD = M)

29 has the value $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ and the result of UNPACK (V, MASK = Q, FIELD = 0) has the value

30 $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$.

31 **13.7.181 VERIFY (STRING, SET [, BACK, KIND])**

32 **Description.** Verify that a set of characters contains all the characters in a string by identifying the
 33 position of the first character in a string of characters that does not appear in a given set of characters.

1 **Class.** Elemental function.

2 **Arguments.**

3 **STRING** shall be of type character.

4 **SET** shall be of type character with the same kind type parameter as **STRING**.

5 **BACK** (optional) shall be of type logical.

6 **KIND** (optional) shall be a scalar integer initialization expression.

7 **Result Characteristics.** Integer. If **KIND** is present, the kind type parameter is that specified by the
8 value of **KIND**; otherwise the kind type parameter is that of default integer type.

9 **Result Value.**

10 *Case (i):* If **BACK** is absent or has the value false and if **STRING** contains at least one character
11 that is not in **SET**, the value of the result is the position of the leftmost character of
12 **STRING** that is not in **SET**.

13 *Case (ii):* If **BACK** is present with the value true and if **STRING** contains at least one character
14 that is not in **SET**, the value of the result is the position of the rightmost character of
15 **STRING** that is not in **SET**.

16 *Case (iii):* The value of the result is zero if each character in **STRING** is in **SET** or if **STRING** has
17 zero length.

18 **Examples.**

19 *Case (i):* `VERIFY ('ABBA', 'A')` has the value 2.

20 *Case (ii):* `VERIFY ('ABBA', 'A', BACK = .TRUE.)` has the value 3.

21 *Case (iii):* `VERIFY ('ABBA', 'AB')` has the value 0.

22 13.8 Standard intrinsic modules

23 13.8.1 General

24 This part of ISO/IEC 1539 defines five standard intrinsic modules: a Fortran environment module, a
25 set of three modules to support exception handling and IEEE arithmetic, and a module to support
26 interoperability with the C programming language.

27 The `IEEE_EXCEPTIONS`, `IEEE_ARITHMETIC`, and `IEEE_FEATURES` intrinsic modules are describ-
28 ed in Clause 14. The `ISO_C_BINDING` intrinsic module is described in Clause 15.

29 A processor may extend the standard intrinsic modules to provide public entities in them in addition to
30 those specified in this part of ISO/IEC 1539.

NOTE 13.24

To avoid potential name conflicts with program entities, it is recommended that a program use the **ONLY** option in any **USE** statement that references a standard intrinsic module.

31 13.8.2 The `ISO_FORTRAN_ENV` intrinsic module

32 13.8.2.1 General

The intrinsic module `ISO_FORTRAN_ENV` provides public entities relating to the Fortran environment.

1 The processor shall provide the named constants, derived type, and procedures described in subclause
2 13.8.2.

3 13.8.2.2 CHARACTER_STORAGE_SIZE

4 The value of the default integer scalar constant CHARACTER_STORAGE_SIZE is the size expressed
5 in bits of the character storage unit (16.5.3.2).

6 13.8.2.3 COMPILER_OPTIONS ()

7 **Description.** Returns a processor-dependent string describing the options that controlled the program
8 translation phase.

9 **Class.** Inquiry function.

10 **Argument.** None.

11 **Result Characteristics.** Default character scalar with processor-dependent length.

12 **Result Value.** A processor-dependent value which describes the options that controlled the translation
13 phase of program execution.

14 **Example.** COMPILER_OPTIONS () might have the value '/OPTIMIZE /FLOAT=IEEE'.

15 13.8.2.4 COMPILER_VERSION ()

16 **Description.** Returns a processor-dependent string identifying the program translation phase.

17 **Class.** Inquiry function.

18 **Argument.** None.

19 **Result Characteristics.** Default character scalar with processor-dependent length.

20 **Result Value.** A processor-dependent value that identifies the name and version of the program
21 translation phase of the processor.

22 **Example.** COMPILER_VERSION () might have the value 'Fast KL-10 Compiler Version 7'.

NOTE 13.25

For both COMPILER_OPTIONS and COMPILER_VERSION the processor should include relevant information that could be useful in solving problems found long after the translation phase. For example, compiler release and patch level, default compiler arguments, environment variable values, and run time library requirements might be included. 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.

23 13.8.2.5 ERROR_UNIT

24 The value of the default integer scalar constant ERROR_UNIT identifies the processor-dependent pre-
25 connected external unit used for the purpose of error reporting (9.4). This unit may be the same as
26 OUTPUT_UNIT. The value shall not be -1.

27 13.8.2.6 FILE_STORAGE_SIZE

- 1 The value of the default integer scalar constant `FILE_STORAGE_SIZE` is the size expressed in bits of
 2 the file storage unit (9.2.4).

3 13.8.2.7 IMAGE_TEAM

- 4 A scalar object of type `IMAGE_TEAM` identifies a team of images. This type is extensible, has only
 5 private components, has no co-array components, and has default initialization to a value that identifies
 6 an empty image set.

J3 internal note

Unresolved Technical Issue 090

Since the author of 06-322r4 claimed he intended `IMAGE_TEAM` to be extensible, I added an edit to that effect. I wonder if that is what he meant?

Does it have pointer components?

Does it have allocatable components?

Even should we ignore or fudge the image accessibility issue, the answer re allocatable components affects what can be done with it on a single image. (Consider `INTENT(OUT)` assumed-size arrays.)

See also UTI 073.

NOTE 13.26

When values of type `IMAGE_TEAM` are constructed by calling the intrinsic subroutine `FORM_TEAM` on the images of a team, the processor may choose to store information in such values to speed later processing of team synchronizations and collective subroutine calls. Since this information is likely to vary between images, it is inappropriate to copy values of type `IMAGE_TEAM` between images.

J3 internal note

Unresolved Technical Issue 073

`IMAGE_TEAM` is still not adequately explained. Better, but not there yet.

Nor are its restrictions.

I said: “Yes, there are certainly valid cross-image uses. Some of these are even obvious (diagnostic information for example).”

Paper 06-322r4 claimed that this did not restrict “the programmer in any significant way”. Well, views might differ on that.

But the explanation is still not satisfactory, and on further reflection, I believe there is no need for such coyness and special molly-coddling of this type.

In other words, I suggest that the special restrictions on `IMAGE_TEAM` be simply deleted. Going by the rationale given by 06-322r4, we can simply say that it *has pointer components*. Then the “pointer ultimate component” restrictions automatically kick in.

NOTE: This of course does not tie the hands of any implementation in any way whatsoever, since the user cannot validly do anything to a private pointer component. The author of 06-322r4 said he wants to allow pointer components without realising that just saying it has them does everything he wants!

So, I suggest:

1. say that `IMAGE_TEAM` has pointer components;
2. delete the special restrictions;
3. delete the note(s?) explaining the special restrictions.

7 13.8.2.8 INPUT_UNIT

1 The value of the default integer scalar constant INPUT_UNIT identifies the same processor-dependent
2 preconnected external unit as the one identified by an asterisk in a READ statement (9.4). The value
3 shall not be -1 .

4 **13.8.2.9 INT8, INT16, INT32, and INT64**

5 The values of these default integer scalar named constants shall be those of the kind type parameters that
6 specify an INTEGER type whose storage size expressed in bits is 8, 16, 32, and 64 respectively. If, for
7 any of these constants, the processor supports more than one kind of that size, it is processor-dependent
8 which kind value is provided. If the processor supports no kind of a particular size, that constant shall
9 be equal to -2 if the processor supports kinds of a larger size and -1 otherwise.

10 **13.8.2.10 IOSTAT_END**

11 The value of the default integer scalar constant IOSTAT_END is assigned to the variable specified in
12 an IOSTAT= specifier (9.10.4) if an end-of-file condition occurs during execution of an input/output
13 statement and no error condition occurs. This value shall be negative.

14 **13.8.2.11 IOSTAT_EOR**

15 The value of the default integer scalar constant IOSTAT_EOR is assigned to the variable specified in
16 an IOSTAT= specifier (9.10.4) if an end-of-record condition occurs during execution of an input/output
17 statement and no end-of-file or error condition occurs. This value shall be negative and different from
18 the value of IOSTAT_END.

19 **13.8.2.12 IOSTAT_INQUIRE_INTERNAL_UNIT**

20 The value of the default integer scalar constant IOSTAT_INQUIRE_INTERNAL_UNIT is assigned to
21 the variable specified in an IOSTAT= specifier in an INQUIRE statement (9.9) if a *file-unit-number*
22 identifies an internal unit in that statement.

NOTE 13.27

This can only occur when a user defined derived type input/output procedure is called by the processor as the result of executing a parent data transfer statement for an internal unit.

23 **13.8.2.13 NUMERIC_STORAGE_SIZE**

24 The value of the default integer scalar constant NUMERIC_STORAGE_SIZE is the size expressed in
25 bits of the numeric storage unit (16.5.3.2).

26 **13.8.2.14 OUTPUT_UNIT**

27 The value of the default integer scalar constant OUTPUT_UNIT identifies the same processor-dependent
28 preconnected external unit as the one identified by an asterisk in a WRITE statement (9.4). The value
29 shall not be -1 .

30 **13.8.2.15 REAL32, REAL64, and REAL128**

31 The values of these default integer scalar named constants shall be those of the kind type parameters
32 that specify a REAL type whose storage size expressed in bits is 32, 64, and 128 respectively. If, for
33 any of these constants, the processor supports more than one kind of that size, it is processor-dependent
34 which kind value is provided. If the processor supports no kind of a particular size, that constant shall
35 be equal to -2 if the processor supports kinds of a larger size and -1 otherwise.

14 Exceptions and IEEE arithmetic

The intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES provide support for exceptions and IEEE arithmetic. Whether the modules are provided is processor dependent. If the module IEEE_FEATURES is provided, which of the named constants defined in this part of ISO/IEC 1539 are included is processor dependent. The module IEEE_ARITHMETIC behaves as if it contained a USE statement for IEEE_EXCEPTIONS; everything that is public in IEEE_EXCEPTIONS is public in IEEE_ARITHMETIC.

NOTE 14.1

The types and procedures defined in these modules are not themselves intrinsic.

If IEEE_EXCEPTIONS or IEEE_ARITHMETIC is accessible in a scoping unit, IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are supported in the scoping unit for all kinds of real and complex data. Which other exceptions are supported can be determined by the function IEEE_SUPPORT_FLAG (14.10.26); whether control of halting is supported can be determined by the function IEEE_SUPPORT_HALTING. The extent of support of the other exceptions may be influenced by the accessibility of the named constants IEEE_INEXACT_FLAG, IEEE_INVALID_FLAG, and IEEE_UNDERFLOW_FLAG of the module IEEE_FEATURES. If a scoping unit has access to IEEE_UNDERFLOW_FLAG of IEEE_FEATURES, within the scoping unit the processor shall support underflow and return true from IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW, X) for at least one kind of real. Similarly, if IEEE_INEXACT_FLAG or IEEE_INVALID_FLAG is accessible, within the scoping unit the processor shall support the exception and return true from the corresponding inquiry for at least one kind of real. Also, if IEEE_HALTING is accessible, within the scoping unit the processor shall support control of halting and return true from IEEE_SUPPORT_HALTING(FLAG) for the flag.

NOTE 14.2

IEEE_INVALID is not required to be supported whenever IEEE_EXCEPTIONS is accessed. This is to allow a non-IEEE processor to provide support for overflow and divide_by_zero. On an IEEE machine, invalid is an equally serious condition.

NOTE 14.3

The IEEE_FEATURES module is provided to allow a reasonable amount of cooperation between the programmer and the processor in controlling the extent of IEEE arithmetic support. On some processors some IEEE features are natural for the processor to support, others may be inefficient at run time, and others are essentially impossible to support. If IEEE_FEATURES is not used, the processor will support only the natural operations. Within IEEE_FEATURES the processor will define the named constants (14.1) corresponding to the time-consuming features (as well as the natural ones for completeness) but will not define named constants corresponding to the impossible features. If the programmer accesses IEEE_FEATURES, the processor shall provide support for all of the IEEE_FEATURES that are reasonably possible. If the programmer uses an ONLY option on a USE statement to access a particular feature name, the processor shall provide support for the corresponding feature, or issue an error message saying the name is not defined in the module.

When used this way, the named constants in the IEEE_FEATURES are similar to what are frequently called command line switches for the compiler. They can specify compilation options in a reasonably portable manner.

1 If a scoping unit does not access IEEE_FEATURES, IEEE_EXCEPTIONS, or IEEE_ARITHMETIC,
 2 the level of support is processor dependent, and need not include support for any exceptions. If a flag is
 3 signaling on entry to such a scoping unit, the processor ensures that it is signaling on exit. If a flag is
 4 quiet on entry to such a scoping unit, whether it is signaling on exit is processor dependent.

5 Further IEEE support is available through the module IEEE_ARITHMETIC. The extent of support may
 6 be influenced by the accessibility of the named constants of the module IEEE_FEATURES. If a scoping
 7 unit has access to IEEE_DATATYPE of IEEE_FEATURES, within the scoping unit the processor shall
 8 support IEEE arithmetic and return true from IEEE_SUPPORT_DATATYPE(X) (14.10.23) for at least
 9 one kind of real. Similarly, if IEEE_DENORMAL, IEEE_DIVIDE, IEEE_INF, IEEE_NAN, IEEE_-
 10 ROUNDING, or IEEE_SQRT is accessible, within the scoping unit the processor shall support the feature
 11 and return true from the corresponding inquiry function for at least one kind of real. In the case of
 12 IEEE_ROUNDING, it shall return true for all the rounding modes IEEE_NEAREST, IEEE_TO_ZERO,
 13 IEEE_UP, and IEEE_DOWN.

14 Execution might be slowed on some processors by the support of some features. If IEEE_EXCEPTIONS
 15 or IEEE_ARITHMETIC is accessed but IEEE_FEATURES is not accessed, the supported subset of fea-
 16 tures is processor dependent. The processor's fullest support is provided when all of IEEE_FEATURES
 17 is accessed as in

18 USE, INTRINSIC :: IEEE_ARITHMETIC; USE, INTRINSIC :: IEEE_FEATURES

19 but execution might then be slowed by the presence of a feature that is not needed. In all cases, the
 20 extent of support can be determined by the inquiry functions.

21 14.1 Derived types and constants defined in the modules

22 The modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES define five derived
 23 types, whose components are all private. No direct component of any of these types is allocatable or a
 24 pointer.

J3 internal note

Unresolved Technical Issue 098

Do they have allocatable components? I would guess NO.

Do they have pointer components? I would guess NO.

These have impact in F2008, because they affect cross-image accessibility.

The text above reflects the editor's attempt to resolve the problem. If it is found to be correct on review, all that is needed is to delete this UTI (but see below).

Note: The first one has some impact in F2003 as well, because it affects INTENT(OUT) assumed-size arrays. That is a defect in F2003. But it gets bigger in F2008. An interp is probably warranted.

This also affects TYPE(C_PTR) and TYPE(C_FUNPTR); the text the editor added to these (see 15.3.3) also needs to be reviewed and accepted before deleting this UTI.

25 The module IEEE_EXCEPTIONS defines the following types.

- 26 • IEEE_FLAG_TYPE is for identifying a particular exception flag. Its only possible values are those
 27 of named constants defined in the module: IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_-
 28 BY_ZERO, IEEE_UNDERFLOW, and IEEE_INEXACT. The module also defines the array named
 29 constants IEEE_USUAL = [IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_INVALID]
 30 and IEEE_ALL = [IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT].
- IEEE_STATUS_TYPE is for containing the floating-point status.

1 The module IEEE_ARITHMETIC defines the following.

- 2 • The type IEEE_CLASS_TYPE, for identifying a class of floating-point values. Its only possible
3 values are those of named constants defined in the module: IEEE_SIGNALING_NAN, IEEE_QUI-
4 ET_NAN, IEEE_NEGATIVE_INF, IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_DENORM-
5 AL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, IEEE_POSITIVE_DENORMAL, IEEE_-
6 POSITIVE_NORMAL, IEEE_POSITIVE_INF, and IEEE_OTHER_VALUE.
- 7 • The type IEEE_ROUND_TYPE, for identifying a particular rounding mode. Its only possible
8 values are those of named constants defined in the module: IEEE_NEAREST, IEEE_TO_ZERO,
9 IEEE_UP, and IEEE_DOWN for the IEEE modes; and IEEE_OTHER for any other mode.
- 10 • The elemental operator == for two values of one of these types to return true if the values are the
11 same and false otherwise.
- 12 • The elemental operator /= for two values of one of these types to return true if the values differ
13 and false otherwise.

14 The module IEEE_FEATURES defines the type IEEE_FEATURES_TYPE, for expressing the need
15 for particular IEEE features. Its only possible values are those of named constants defined in the
16 module: IEEE_DATATYPE, IEEE_DENORMAL, IEEE_DIVIDE, IEEE_HALTING, IEEE_INEXACT_-
17 FLAG, IEEE_INF, IEEE_INVALID_FLAG, IEEE_NAN, IEEE_ROUNDING, IEEE_SQRT, and IEEE_-
18 UNDERFLOW_FLAG.

19 14.2 The exceptions

20 The exceptions are the following.

- 21 • IEEE_OVERFLOW occurs when the result for an intrinsic real operation or assignment has an
22 absolute value greater than a processor-dependent limit, or the real or imaginary part of the result
23 for an intrinsic complex operation or assignment has an absolute value greater than a processor-
24 dependent limit.
- 25 • IEEE_DIVIDE_BY_ZERO occurs when a real or complex division has a nonzero numerator and a
26 zero denominator.
- 27 • IEEE_INVALID occurs when a real or complex operation or assignment is invalid; possible examples
28 are SQRT(X) when X is real and has a nonzero negative value, and conversion to an integer (by
29 assignment, an intrinsic procedure, or a procedure defined in an intrinsic module) when the result
30 is too large to be representable.
- 31 • IEEE_UNDERFLOW occurs when the result for an intrinsic real operation or assignment has an
32 absolute value less than a processor-dependent limit and loss of accuracy is detected, or the real
33 or imaginary part of the result for an intrinsic complex operation or assignment has an absolute
34 value less than a processor-dependent limit and loss of accuracy is detected.
- 35 • IEEE_INEXACT occurs when the result of a real or complex operation or assignment is not exact.

36 Each exception has a flag whose value is either quiet or signaling. The value can be determined by
37 the function IEEE_GET_FLAG. Its initial value is quiet and it signals when the associated exception
38 occurs. Its status can also be changed by the subroutine IEEE_SET_FLAG or the subroutine IEEE_-
39 SET_STATUS. Once signaling within a procedure, it remains signaling unless set quiet by an invocation
40 of the subroutine IEEE_SET_FLAG or the subroutine IEEE_SET_STATUS.

- 1 If a flag is signaling on entry to a procedure other than IEEE.GET_FLAG or IEEE.GET_STATUS, the
 2 processor will set it to quiet on entry and restore it to signaling on return.

NOTE 14.4

If a flag signals during execution of a procedure, the processor shall not set it to quiet on return.

- 3 Evaluation of a specification expression might cause an exception to signal.

4 In a scoping unit that has access to IEEE.EXCEPTIONS or IEEE.ARITHMETIC, if an intrinsic pro-
 5 cedure or a procedure defined in an intrinsic module executes normally, the values of the flags IEEE-
 6 OVERFLOW, IEEE.DIVIDE_BY_ZERO, and IEEE.INVALID shall be as on entry to the procedure,
 7 even if one or more signals during the calculation. If a real or complex result is too large for the pro-
 8 cedure to handle, IEEE.OVERFLOW may signal. If a real or complex result is a NaN because of an
 9 invalid operation (for example, LOG(-1.0)), IEEE.INVALID may signal. Similar rules apply to format
 10 processing and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set
 11 signaling because of an intermediate calculation that does not affect the result.

12 In a sequence of statements that has no invocations of IEEE.GET_FLAG, IEEE.SET_FLAG, IEEE-
 13 GET_STATUS, IEEE.SET_HALTING_MODE, or IEEE.SET_STATUS, if the execution of an operation
 14 would cause an exception to signal but after execution of the sequence no value of a variable depends
 15 on the operation, whether the exception is signaling is processor dependent. For example, when Y has
 16 the value zero, whether the code

```
17      X = 1.0/Y
18      X = 3.0
```

- 19 signals IEEE.DIVIDE_BY_ZERO is processor dependent. Another example is the following:

```
20      REAL, PARAMETER :: X=0.0, Y=6.0
21      IF (1.0/X == Y) PRINT *, 'Hello world'
```

22 where the processor is permitted to discard the IF statement because the logical expression can never
 23 be true and no value of a variable depends on it.

24 An exception shall not signal if this could arise only during execution of an operation beyond those
 25 required or permitted by the standard. For example, the statement

```
26      IF (F(X)>0.0) Y = 1.0/Z
```

- 27 shall not signal IEEE.DIVIDE_BY_ZERO when both F(X) and Z are zero and the statement

```
28      WHERE(A>0.0) A = 1.0/A
```

29 shall not signal IEEE.DIVIDE_BY_ZERO. On the other hand, when X has the value 1.0 and Y has the
 30 value 0.0, the expression

```
31      X>0.00001 .OR. X/Y>0.00001
```

- 1 is permitted to cause the signaling of IEEE_DIVIDE_BY_ZERO.
- 2 The processor need not support IEEE_INVALID, IEEE_UNDERFLOW, and IEEE_INEXACT. If an
3 exception is not supported, its flag is always quiet. The function IEEE_SUPPORT_FLAG can be used
4 to inquire whether a particular flag is supported.

5 **14.3 The rounding modes**

6 The IEEE International Standard specifies four rounding modes.

- 7 • IEEE_NEAREST rounds the exact result to the nearest representable value.
- 8 • IEEE_TO_ZERO rounds the exact result towards zero to the next representable value.
- 9 • IEEE_UP rounds the exact result towards +infinity to the next representable value.
- 10 • IEEE_DOWN rounds the exact result towards -infinity to the next representable value.

11 The function IEEE_GET_ROUNDING_MODE can be used to inquire which rounding mode is in opera-
12 tion. Its value is one of the above four or IEEE_OTHER if the rounding mode does not conform to the
13 IEEE International Standard.

14 If the processor supports the alteration of the rounding mode during execution, the subroutine IEEE_-
15 SET_ROUNDING_MODE can be used to alter it. The function IEEE_SUPPORT_ROUNDING can be
16 used to inquire whether this facility is available for a particular mode. The function IEEE_SUPPORT_-
17 IO can be used to inquire whether rounding for base conversion in formatted input/output (9.4.5.14,
18 9.5.2.13, 10.7.2.3.7) is as specified in the IEEE International Standard.

19 In a procedure other than IEEE_SET_ROUNDING_MODE or IEEE_SET_STATUS, the processor shall
20 not change the rounding mode on entry, and on return shall ensure that the rounding mode is the same
21 as it was on entry.

NOTE 14.5

Within a program, all literal constants that have the same form have the same value (4.1.2).
Therefore, the value of a literal constant is not affected by the rounding mode.

22 **14.4 Underflow mode**

23 Some processors allow control during program execution of whether underflow produces a denormalized
24 number in conformance with the IEEE International Standard (gradual underflow) or produces zero
25 instead (abrupt underflow). On some processors, floating-point performance is typically better in abrupt
26 underflow mode than in gradual underflow mode.

27 Control over the underflow mode is exercised by invocation of IEEE_SET_UNDERFLOW_MODE. The
28 function IEEE_GET_UNDERFLOW_MODE can be used to inquire which underflow mode is in opera-
29 tion. The function IEEE_SUPPORT_UNDERFLOW_MODE can be used to inquire whether this facility
30 is available. The initial underflow mode is processor dependent. In a procedure other than IEEE_SET_-
31 UNDERFLOW_MODE or IEEE_SET_STATUS, the processor shall not change the underflow mode on
32 entry, and on return shall ensure that the underflow mode is the same as it was on entry.

33 The underflow mode affects only floating-point calculations whose type is that of an X for which IEEE_-
34 SUPPORT_UNDERFLOW_CONTROL returns true.

14.5 Halting

Some processors allow control during program execution of whether to abort or continue execution after an exception. Such control is exercised by invocation of the subroutine `IEEE.SET_HALTING_MODE`. Halting is not precise and may occur any time after the exception has occurred. The function `IEEE.SUPPORT_HALTING` can be used to inquire whether this facility is available. The initial halting mode is processor dependent. In a procedure other than `IEEE.SET_HALTING_MODE` or `IEEE.SET_STATUS`, the processor shall not change the halting mode on entry, and on return shall ensure that the halting mode is the same as it was on entry.

14.6 The floating-point status

The values of all the supported flags for exceptions, rounding mode, underflow mode, and halting are called the floating-point status. The floating-point status can be saved in a scalar variable of type `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 held within such a variable. Portions of the floating-point status can be saved with the subroutines `IEEE.GET_FLAG`, `IEEE.GET_HALTING_MODE`, and `IEEE.GET_ROUNDING_MODE`, and set with the subroutines `IEEE.SET_FLAG`, `IEEE.SET_HALTING_MODE`, and `IEEE.SET_ROUNDING_MODE`.

NOTE 14.6

Some processors hold all these flags in a floating-point status register that can be saved and restored as a whole much faster than all individual flags can be saved and restored. These procedures are provided to exploit this feature.

NOTE 14.7

The processor is required to ensure that a call to a Fortran procedure does not change the floating-point status other than by setting exception flags to signaling.

14.7 Exceptional values

The IEEE International Standard specifies the following exceptional floating-point values.

- Denormalized values have very small absolute values and lowered 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.

In this part of ISO/IEC 1539, the term **normal** is used for values that are not in one of these exceptional classes.

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 functions `IEEE.SUPPORT_DENORMAL`, `IEEE.SUPPORT_INF`, and `IEEE.SUPPORT_NAN` are provided to determine whether these facilities are available for a particular kind of real.

14.8 IEEE arithmetic

The function `IEEE_SUPPORT_DATATYPE` can be used to inquire whether IEEE arithmetic is supported for a particular kind of real. Complete conformance with the IEEE International Standard is not required, but the normalized numbers shall be exactly those of an IEEE floating-point format; the operations of addition, subtraction, and multiplication shall conform with at least one of the IEEE rounding modes; the IEEE operation `rem` shall be provided by the function `IEEE_REM`; and the IEEE functions `copysign`, `scalb`, `logb`, `nextafter`, and `unordered` shall be provided by the functions `IEEE_COPY_SIGN`, `IEEE_SCALB`, `IEEE_LOGB`, `IEEE_NEXT_AFTER`, and `IEEE_UNORDERED`, respectively. The inquiry function `IEEE_SUPPORT_DIVIDE` is provided to inquire whether the processor supports divide with the accuracy specified by the IEEE International Standard. For each of the operations of addition, subtraction, and multiplication, the result shall be as specified in the IEEE International Standard whenever the IEEE result is normalized and the operands are normalized (if floating-point) or are valid and within range (if another type).

The inquiry function `IEEE_SUPPORT_NAN` is provided to inquire whether the processor supports IEEE NaNs. Where these are supported, their behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifications in the IEEE International Standard.

The inquiry function `IEEE_SUPPORT_INF` is provided to inquire whether the processor supports IEEE infinities. Where these are supported, their behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifications in the IEEE International Standard.

The inquiry function `IEEE_SUPPORT_DENORMAL` is provided to inquire whether the processor supports IEEE denormals. Where these are supported, their behavior for unary and binary operations, including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifications in the IEEE International Standard.

The IEEE International Standard specifies a square root function that returns -0.0 for the square root of -0.0 and has certain accuracy requirements. The function `IEEE_SUPPORT_SQRT` can be used to inquire whether `SQRT` conforms to the IEEE International Standard for a particular kind of real.

The inquiry function `IEEE_SUPPORT_STANDARD` is provided to inquire whether the processor supports all the IEEE facilities defined in this part of ISO/IEC 1539 for a particular kind of real.

14.9 Tables 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.

The procedure classification terms “inquiry function” and “transformational function” are used here with the same meanings as in 13.1.

14.9.1 Inquiry functions

The module `IEEE_EXCEPTIONS` contains the following inquiry functions.

<code>IEEE_SUPPORT_FLAG</code> (<code>FLAG</code> [, <code>X</code>])	Are IEEE exceptions supported?
<code>IEEE_SUPPORT_HALTING</code> (<code>FLAG</code>)	Is IEEE halting control supported?

The module `IEEE_ARITHMETIC` contains the following inquiry functions.

<code>IEEE_SUPPORT_DATATYPE</code> ([<code>X</code>])	Is IEEE arithmetic supported?
<code>IEEE_SUPPORT_DENORMAL</code> ([<code>X</code>])	Are IEEE denormalized numbers supported?
<code>IEEE_SUPPORT_DIVIDE</code> ([<code>X</code>])	Is IEEE divide supported?

1	IEEE_SUPPORT_INF ([X])	Is IEEE infinity supported?
2	IEEE_SUPPORT_IO ([X])	Is IEEE formatting supported?
3	IEEE_SUPPORT_NAN ([X])	Are IEEE NaNs supported?
	IEEE_SUPPORT_ROUNDING	Is IEEE rounding supported?
4	(ROUND_VALUE [, X])	
5	IEEE_SUPPORT_SQRT ([X])	Is IEEE square root supported?
6	IEEE_SUPPORT_STANDARD ([X])	Are all IEEE facilities supported?
	IEEE_SUPPORT_UNDERFLOW_-	Is IEEE underflow control supported?
	CONTROL ([X])	

8 14.9.2 Elemental functions

9 The module IEEE_ARITHMETIC contains the following elemental functions for reals X and Y for which
10 IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_DATATYPE(Y) are true.

11	IEEE_CLASS (X)	IEEE class.
12	IEEE_COPY_SIGN (X,Y)	IEEE copysign function.
13	IEEE_IS_FINITE (X)	Determine if value is finite.
14	IEEE_IS_NAN (X)	Determine if value is IEEE Not-a-Number.
15	IEEE_IS_NORMAL (X)	Determine if a value is normal, that is, neither an infinity, a NaN, nor denormalized.
16	IEEE_IS_NEGATIVE (X)	Determine if value is negative.
17	IEEE_LOGB (X)	Unbiased exponent in the IEEE floating-point format.
18		
19	IEEE_NEXT_AFTER (X,Y)	Returns the next representable neighbor of X in the direction toward Y.
20		
21	IEEE_REM (X,Y)	The IEEE REM function, that is $X - Y*N$, where N is the integer nearest to the exact value X/Y.
22		
23		
24	IEEE_RINT (X)	Round to an integer value according to the current rounding mode.
25		
26	IEEE_SCALB (X,I)	Returns $X \times 2^I$.
27	IEEE_UNORDERED (X,Y)	IEEE unordered function. True if X or Y is a NaN and false otherwise.
28		
29		
30	IEEE_VALUE (X,CLASS)	Generate an IEEE value.

31 14.9.3 Kind function

32 The module IEEE_ARITHMETIC contains the following transformational function.

33	IEEE_SELECTED_REAL_KIND ([P, R,	Kind type parameter value for an IEEE real with
34	RADIX])	given precision, range, and radix.

35 14.9.4 Elemental subroutines

36 The module IEEE_EXCEPTIONS contains the following elemental subroutines.

37	IEEE_GET_FLAG (FLAG,FLAG_VALUE)	Get an exception flag.
38	IEEE_GET_HALTING_MODE (FLAG,	Get halting mode for an exception.
	HALTING)	

39 14.9.5 Nonelemental subroutines

1 The module IEEE_EXCEPTIONS contains the following nonelemental subroutines.

2	IEEE_GET_STATUS (STATUS_VALUE)	Get the current state of the floating-point
3		environment.
4	IEEE_SET_FLAG (FLAG, FLAG_VALUE)	Set an exception flag.
	IEEE_SET_HALTING_MODE (FLAG,	Controls continuation or halting on exceptions.
5	HALTING)	
6	IEEE_SET_STATUS (STATUS_VALUE)	Restore the state of the floating-point
7		environment.

8 The nonelemental subroutines IEEE_SET_FLAG and IEEE_SET_HALTING_MODE are pure. No other
9 nonelemental subroutine contained in IEEE_EXCEPTIONS is pure.

10 The module IEEE_ARITHMETIC contains the following nonelemental subroutines.

	IEEE_GET_ROUNDING_MODE	Get the current IEEE rounding mode.
11	(ROUND_VALUE)	
	IEEE_GET_UNDERFLOW_MODE	Get the current underflow mode.
12	(GRADUAL)	
	IEEE_SET_ROUNDING_MODE	Set the current IEEE rounding mode.
13	(ROUND_VALUE)	
	IEEE_SET_UNDERFLOW_MODE	Set the current underflow mode.
14	(GRADUAL)	

15 No nonelemental subroutine contained in IEEE_ARITHMETIC is pure.

16 14.10 Specifications of the procedures

17 In the detailed descriptions below, procedure names are generic and are not specific. All the functions
18 are pure. The dummy arguments of the intrinsic module procedures in 14.9.1, 14.9.2, and 14.9.3 have
19 INTENT(IN). The dummy arguments of the intrinsic module procedures in 14.9.4 and 14.9.5 have
20 INTENT(IN) if the intent is not stated explicitly. In the examples, it is assumed that the processor
21 supports IEEE arithmetic for default real.

NOTE 14.8

It is intended that a processor should not check a condition given in a paragraph labeled
“**Restriction**” at compile time, but rather should rely on the programmer writing code such
as

```

      IF (IEEE_SUPPORT_DATATYPE(X)) THEN
        C = IEEE_CLASS(X)
      ELSE
        .
        .
      ENDIF

```

to avoid a call being made on a processor for which the condition is violated.

22 For the elemental functions of IEEE_ARITHMETIC, as tabulated in 14.9.2, if X or Y has a value that
23 is an infinity or a NaN, the result shall be consistent with the general rules in 6.1 and 6.2 of the IEEE
24 International Standard. For example, the result for an infinity shall be constructed as the limiting case
25 of the result with a value of arbitrarily large magnitude, if such a limit exists.

1 14.10.1 IEEE_CLASS (X)

2 **Description.** IEEE class function.

3 **Class.** Elemental function.

4 **Argument.** X shall be of type real.

5 **Restriction.** IEEE_CLASS(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value
6 false.

7 **Result Characteristics.** TYPE(IEEE_CLASS_TYPE).

8 **Result Value.** The result value shall be IEEE_SIGNALING_NAN or IEEE_QUIET_NAN if IEEE_
9 SUPPORT_NAN(X) has the value true and the value of X is a signaling or quiet NaN, respectively. The
10 result value shall be IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF if IEEE_SUPPORT_INF(X) has
11 the value true and the value of X is negative or positive infinity, respectively. The result value shall
12 be IEEE_NEGATIVE_DENORMAL or IEEE_POSITIVE_DENORMAL if IEEE_SUPPORT_DENOR-
13 MAL(X) has the value true and the value of X is a negative or positive denormalized value, respectively.
14 The result value shall be IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_
15 ZERO, or IEEE_POSITIVE_NORMAL if value of X is negative normal, negative zero, positive zero, or
16 positive normal, respectively. Otherwise, the result value shall be IEEE_OTHER_VALUE.

17 **Example.** IEEE_CLASS(-1.0) has the value IEEE_NEGATIVE_NORMAL.

NOTE 14.9

The result value IEEE_OTHER_VALUE is needed for implementing the module on systems which are basically IEEE, but do not implement all of it. It might be needed, for example, if an unformatted file were written in a program executing with gradual underflow enabled and read with it disabled.

18 14.10.2 IEEE_COPY_SIGN (X, Y)

19 **Description.** IEEE copysign function.

20 **Class.** Elemental function.

21 **Arguments.** The arguments shall be of type real.

22 **Restriction.** IEEE_COPY_SIGN(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or
23 IEEE_SUPPORT_DATATYPE(Y) has the value false.

24 **Result Characteristics.** Same as X.

25 **Result Value.** The result has the value of X with the sign of Y. This is true even for IEEE special
26 values, such as a NaN or an infinity (on processors supporting such values).

27 **Example.** The value of IEEE_COPY_SIGN(X,1.0) is ABS(X) even when X is NaN.

28 14.10.3 IEEE_GET_FLAG (FLAG, FLAG_VALUE)

29 **Description.** Get an exception flag.

30 **Class.** Elemental subroutine.

31 **Arguments.**

FLAG shall be of type TYPE(IEEE_FLAG_TYPE). It specifies the IEEE flag to be obtained.

FLAG_VALUE shall be of type default 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, the result value is true if the corresponding exception flag is signaling and is 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.

14.10.4 IEEE_GET_HALTING_MODE (FLAG, HALTING)

Description. Get halting mode for an exception.

Class. Elemental subroutine.

Arguments.

FLAG shall be of type TYPE(IEEE_FLAG_TYPE). It specifies the IEEE 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 default logical. It is of INTENT(OUT). The value is true if the exception specified by FLAG will cause halting. Otherwise, the value is 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
```

14.10.5 IEEE_GET_ROUNDING_MODE (ROUND_VALUE)

Description. Get the current IEEE rounding mode.

Class. Subroutine.

Argument. ROUND_VALUE shall be scalar of type TYPE(IEEE_ROUND_TYPE). It is an INTENT(OUT) argument and returns the floating-point rounding mode, with value IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, or IEEE_DOWN if one of the IEEE modes is in operation and IEEE_OTHER otherwise.

Example. To store the rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
...
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
```

```

1      ... ! calculation with round to nearest
2      CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode

```

3 14.10.6 IEEE_GET_STATUS (STATUS_VALUE)

4 **Description.** Get the current value of the floating-point status (14.6).

5 **Class.** Subroutine.

6 **Argument.** STATUS_VALUE shall be scalar of type TYPE(IEEE_STATUS_TYPE). It is an IN-
7 TENT(OUT) argument and returns the floating-point status.

8 **Example.** To store all the exception flags, do a calculation involving exception handling, and restore
9 them later:

```

10      USE, INTRINSIC :: IEEE_ARITHMETIC
11      TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
12      ...
13      CALL IEEE_GET_STATUS(STATUS_VALUE) ! Get the flags
14      CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set the flags quiet.
15      ... ! calculation involving exception handling
16      CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags

```

17 14.10.7 IEEE_GET_UNDERFLOW_MODE (GRADUAL)

18 **Description.** Get the current underflow mode (14.4).

19 **Class.** Subroutine.

20 **Argument.** GRADUAL shall be scalar and of type default logical. It is an INTENT(OUT) argument.
21 The value is true if the current underflow mode is gradual underflow, and false if the current underflow
22 mode is abrupt underflow.

23 **Restriction.** IEEE_GET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UN-
24 DERFLOW_CONTROL(X) is true for some X.

25 **Example.** After CALL IEEE.SET.UNDERFLOW_MODE(.FALSE.), a subsequent CALL IEEE.-
26 GET.UNDERFLOW_MODE(GRADUAL) will set GRADUAL to false.

27 14.10.8 IEEE_IS_FINITE (X)

28 **Description.** Determine if a value is finite.

29 **Class.** Elemental function.

30 **Argument.** X shall be of type real.

31 **Restriction.** IEEE_IS_FINITE(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the
32 value false.

33 **Result Characteristics.** Default logical.

34 **Result Value.** The result has the value true if the value of X is finite, that is, IEEE_CLASS(X) has one of
35 the values IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_DENORMAL, IEEE_NEGATIVE_ZERO,

1 IEEE_POSITIVE_ZERO, IEEE_POSITIVE_DENORMAL, or IEEE_POSITIVE_NORMAL; otherwise,
2 the result has the value false.

3 **Example.** IEEE_IS_FINITE(1.0) has the value true.

4 **14.10.9 IEEE_IS_NAN (X)**

5 **Description.** Determine if a value is IEEE Not-a-Number.

6 **Class.** Elemental function.

7 **Argument.** X shall be of type real.

8 **Restriction.** IEEE_IS_NAN(X) shall not be invoked if IEEE_SUPPORT_NAN(X) has the value false.

9 **Result Characteristics.** Default logical.

10 **Result Value.** The result has the value true if the value of X is an IEEE NaN; otherwise, it has the
11 value false.

12 **Example.** IEEE_IS_NAN(SQRT(-1.0)) has the value true if IEEE_SUPPORT_SQRT(1.0) has the value
13 true.

14 **14.10.10 IEEE_IS_NEGATIVE (X)**

15 **Description.** Determine if a value is negative.

16 **Class.** Elemental function.

17 **Argument.** X shall be of type real.

18 **Restriction.** IEEE_IS_NEGATIVE(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has
19 the value false.

20 **Result Characteristics.** Default logical.

21 **Result Value.** The result has the value true if IEEE_CLASS(X) has one of the values IEEE_NEGA-
22 TIVE_NORMAL, IEEE_NEGATIVE_DENORMAL, IEEE_NEGATIVE_ZERO or IEEE_NEGATIVE_-
23 INF; otherwise, the result has the value false.

24 **Example.** IEEE_IS_NEGATIVE(0.0)) has the value false.

25 **14.10.11 IEEE_IS_NORMAL (X)**

26 **Description.** Determine if a value is normal, that is, neither an infinity, a NaN, nor denormalized.

27 **Class.** Elemental function.

28 **Argument.** X shall be of type real.

29 **Restriction.** IEEE_IS_NORMAL(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the
30 value false.

31 **Result Characteristics.** Default logical.

32 **Result Value.** The result has the value true if IEEE_CLASS(X) has one of the values IEEE_NEGA-
33 TIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO or IEEE_POSITIVE_NORMAL;
34 otherwise, the result has the value false.

1 **Example.** IEEE_IS_NORMAL(SQRT(-1.0)) has the value false if IEEE_SUPPORT_SQRT(1.0) has the
2 value true.

3 **14.10.12 IEEE_LOGB (X)**

4 **Description.** Unbiased exponent in the IEEE floating-point format.

5 **Class.** Elemental function.

6 **Argument.** X shall be of type real.

7 **Restriction.** IEEE_LOGB(X) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has the value
8 false.

9 **Result Characteristics.** Same as X.

10 **Result Value.**

11 *Case (i):* If the value of X is neither zero, infinity, nor NaN, the result has the value of the unbiased
12 exponent of X. Note: this value is equal to EXPONENT(X)-1.

13 *Case (ii):* If X==0, the result is -infinity if IEEE_SUPPORT_INF(X) is true and -HUGE(X) other-
14 wise; IEEE_DIVIDE_BY_ZERO signals.

15 **Example.** IEEE_LOGB(-1.1) has the value 0.0.

16 **14.10.13 IEEE_NEXT_AFTER (X, Y)**

17 **Description.** Returns the next representable neighbor of X in the direction toward Y.

18 **Class.** Elemental function.

19 **Arguments.** The arguments shall be of type real.

20 **Restriction.** IEEE_NEXT_AFTER(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or
21 IEEE_SUPPORT_DATATYPE(Y) has the value false.

22 **Result Characteristics.** Same as X.

23 **Result Value.**

24 *Case (i):* If X == Y, the result is X and no exception is signaled.

25 *Case (ii):* If X /= Y, the result has the value of the next representable neighbor of X in the direction
26 of Y. The neighbors of zero (of either sign) are both nonzero. IEEE_OVERFLOW is sig-
27 naled when X is finite but IEEE_NEXT_AFTER(X,Y) is infinite; IEEE_UNDERFLOW
28 is signaled when IEEE_NEXT_AFTER(X,Y) is denormalized; in both cases, IEEE_INEX-
29 ACT signals.

30 **Example.** The value of IEEE_NEXT_AFTER(1.0,2.0) is 1.0+EPSILON(X).

31 **14.10.14 IEEE_REM (X, Y)**

32 **Description.** IEEE REM function.

33 **Class.** Elemental function.

34 **Arguments.** The arguments shall be of type real.

35 **Restriction.** IEEE_REM(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or IEEE_-
SUPPORT_DATATYPE(Y) has the value false.

1

2 **Result Characteristics.** Real with the kind type parameter of whichever argument has the greater
3 precision.

4 **Result Value.** The result value, regardless of the rounding mode, shall be exactly $X - Y * N$, where N
5 is the integer nearest to the exact value X/Y ; whenever $|N - X/Y| = 1/2$, N shall be even. If the result
6 value is zero, the sign shall be that of X .

7 **Examples.** The value of `IEEE_REM(4.0,3.0)` is 1.0, the value of `IEEE_REM(3.0,2.0)` is -1.0, and the
8 value of `IEEE_REM(5.0,2.0)` is 1.0.

9 **14.10.15 IEEE_RINT (X)**

10 **Description.** Round to an integer value according to the current rounding mode.

11 **Class.** Elemental function.

12 **Argument.** X shall be of type real.

13 **Restriction.** `IEEE_RINT(X)` shall not be invoked if `IEEE_SUPPORT_DATATYPE(X)` has the value
14 false.

15 **Result Characteristics.** Same as X .

16 **Result Value.** The value of the result is the value of X rounded to an integer according to the current
17 rounding mode. If the result has the value zero, the sign is that of X .

18 **Examples.** If the current rounding mode is round to nearest, the value of `IEEE_RINT(1.1)` is 1.0. If
19 the current rounding mode is round up, the value of `IEEE_RINT(1.1)` is 2.0.

20 **14.10.16 IEEE_SCALB (X, I)**

21 **Description.** Returns $X \times 2^I$.

22 **Class.** Elemental function.

23 **Arguments.**

24 X shall be of type real.

25 I shall be of type integer.

26 **Restriction.** `IEEE_SCALB(X)` shall not be invoked if `IEEE_SUPPORT_DATATYPE(X)` has the value
27 false.

28 **Result Characteristics.** Same as X .

29 **Result Value.**

30 *Case (i):* If $X \times 2^I$ is representable as a normal number, the result has this value.

31 *Case (ii):* If X is finite and $X \times 2^I$ is too large, the `IEEE_OVERFLOW` exception shall occur. If
32 `IEEE_SUPPORT_INF(X)` is true, the result value is infinity with the sign of X ; otherwise,
33 the result value is `SIGN(HUGE(X),X)`.

34 *Case (iii):* If $X \times 2^I$ is too small and there is loss of accuracy, the `IEEE_UNDERFLOW` exception
35 shall occur. The result is the representable number having a magnitude nearest to $|2^I|$
36 and the same sign as X .

37 *Case (iv):* If X is infinite, the result is the same as X ; no exception signals.

1 **Example.** The value of IEEE_SCALB(1.0,2) is 4.0.

2 **14.10.17 IEEE_SELECTED_REAL_KIND ([P, R, RADIX])**

3 **Description.** Returns a value of the kind type parameter of an IEEE real data type with decimal
4 precision of at least P digits, a decimal exponent range of at least R, and a radix of RADIX. For data
5 objects of such a type, IEEE_SUPPORT_DATATYPE(X) has the value true.

6 **Class.** Transformational function.

7 **Arguments.** At least one argument shall be present.

8 P (optional) shall be scalar and of type integer.

9 R (optional) shall be scalar and of type integer.

10 RADIX (optional) shall be scalar and of type integer.

11 **Result Characteristics.** Default integer scalar.

12 **Result Value.** If P or R is absent, the result value is the same as if it were present with the value zero.
13 If RADIX is absent, there is no requirement on the radix of the selected kind. The result has a value
14 equal to a value of the kind type parameter of an IEEE real type with decimal precision, as returned by
15 the function PRECISION, of at least P digits, a decimal exponent range, as returned by the function
16 RANGE, of at least R, and a radix, as returned by the function RADIX, of RADIX, if such a kind type
17 parameter is available on the processor.

18 Otherwise, the result is -1 if the processor supports an IEEE real type with radix RADIX and exponent
19 range of at least R but not with precision of at least P, -2 if the processor supports an IEEE real
20 type with radix RADIX and precision of at least P but not with exponent range of at least R, -3 if
21 the processor supports an IEEE real type with radix RADIX but with neither precision of at least P
22 nor exponent range of at least R, -4 if the processor supports an IEEE real type with radix RADIX
23 and either precision of at least P or exponent range of at least R but not both together, and -5 if the
24 processor supports no IEEE real type with radix RADIX.

25 If more than one kind type parameter value meets the criteria, the value returned is the one with the
26 smallest decimal precision, unless there are several such values, in which case the smallest of these kind
27 values is returned.

28 **Example.** IEEE_SELECTED_REAL_KIND(6,30) has the value KIND(0.0) on a machine that supports
29 IEEE single precision arithmetic for its default real approximation method.

30 **14.10.18 IEEE_SET_FLAG (FLAG, FLAG_VALUE)**

31 **Description.** Assign a value to an exception flag.

32 **Class.** Pure subroutine.

33 **Arguments.**

34 FLAG shall be a scalar or array of type TYPE(IEEE_FLAG_TYPE). If a value of FLAG
is IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDER-
FLOW, or IEEE_INEXACT, the corresponding exception flag is assigned a value. No
two elements of FLAG shall have the same value.

35 FLAG.VALUE shall be a scalar or array of type default logical. 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.

1 **Example.** CALL IEEE_SET_FLAG(IEEE_OVERFLOW,.TRUE.) sets the IEEE_OVERFLOW flag to
2 be signaling.

3 **14.10.19 IEEE_SET_HALTING_MODE (FLAG, HALTING)**

4 **Description.** Controls continuation or halting after an exception.

5 **Class.** Pure subroutine.

6 **Arguments.**

FLAG shall be a scalar or array of type 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 scalar or array of type default logical. 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.

9 **Restriction.** IEEE_SET_HALTING_MODE(FLAG,HALTING) shall not be invoked if IEEE_SUPPORT_HALTING(FLAG) has the value false.

11 **Example.** CALL IEEE_SET_HALTING_MODE(IEEE_DIVIDE_BY_ZERO,.TRUE.) causes halting after a divide_by_zero exception.

NOTE 14.10

The initial halting mode is processor dependent. Halting is not precise and may occur some time after the exception has occurred.

13 **14.10.20 IEEE_SET_ROUNDING_MODE (ROUND_VALUE)**

14 **Description.** Set the current IEEE rounding mode.

15 **Class.** Subroutine.

16 **Argument.** ROUND_VALUE shall be scalar and of type TYPE(IEEE_ROUND_TYPE). It specifies the mode to be set.

18 **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_DATA_TYPE(X) is true.

21 **Example.** To store the rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```
23      USE, INTRINSIC :: IEEE_ARITHMETIC
24      TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
25      ...
26      CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
27      CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
28      : ! calculation with round to nearest
29      CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode
```

14.10.21 IEEE_SET_STATUS (STATUS_VALUE)

- 1 **Description.** Restore the value of the floating-point status (14.6).
 2 **Class.** Subroutine.
 3 **Argument.** STATUS_VALUE shall be scalar and of type TYPE(IEEE_STATUS_TYPE). Its value shall
 4 have been set in a previous invocation of IEEE_GET_STATUS.
 5 **Example.** To store all the exceptions flags, do a calculation involving exception handling, and restore
 6 them later:

```

7      USE, INTRINSIC :: IEEE_EXCEPTIONS
8      TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
9      ...
10     CALL IEEE_GET_STATUS(STATUS_VALUE) ! Store the flags
11     CALL IEEE_SET_FLAGS(IEEE_ALL,.FALSE.) ! Set them quiet
12     ... ! calculation involving exception handling
13     CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags

```

NOTE 14.11

On some processors this may be a very time consuming process.

14.10.22 IEEE_SET_UNDERFLOW_MODE (GRADUAL)

- 15 **Description.** Set the current underflow mode.
 16 **Class.** Subroutine.
 17 **Argument.** GRADUAL shall be scalar and of type default logical. If it is true, the current underflow
 18 mode is set to gradual underflow. If it is false, the current underflow mode is set to abrupt underflow.
 19 **Restriction.** IEEE_SET_UNDERFLOW_MODE shall not be invoked unless IEEE_SUPPORT_UN-
 20 DERFLOW_CONTROL(X) is true for some X.
 21 **Example.** To perform some calculations with abrupt underflow and then restore the previous mode:

```

22     USE,INTRINSIC :: IEEE_ARITHMETIC
23     LOGICAL SAVE_UNDERFLOW_MODE
24     ...
25     CALL IEEE_GET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)
26     CALL IEEE_SET_UNDERFLOW_MODE(GRADUAL=.FALSE.)
27     ... ! Perform some calculations with abrupt underflow
28     CALL IEEE_SET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)

```

29 14.10.23 IEEE_SUPPORT_DATATYPE () or IEEE_SUPPORT_DATATYPE (X)

- 30 **Description.** Inquire whether the processor supports IEEE arithmetic.
 31 **Class.** Inquiry function.
 32 **Argument.** X shall be of type real. It may be a scalar or an array.

1 **Result Characteristics.** Default logical scalar.

2 **Result Value.** The result has the value true if the processor supports IEEE arithmetic for all reals
 3 (X absent) or for real variables of the same kind type parameter as X; otherwise, it has the value false.
 4 Here, support is as defined in the first paragraph of 14.8.

5 **Example.** If default real type conforms to the IEEE International Standard except that underflow
 6 values flush to zero instead of being denormal, IEEE_SUPPORT_DATATYPE(1.0) has the value true.

7 **14.10.24 IEEE_SUPPORT_DENORMAL () or IEEE_SUPPORT_DENORMAL (X)**

8 **Description.** Inquire whether the processor supports IEEE denormalized numbers.

9 **Class.** Inquiry function.

10 **Argument.** X shall be of type real. It may be a scalar or an array.

11 **Result Characteristics.** Default logical scalar.

12 **Result Value.**

13 *Case (i):* IEEE_SUPPORT_DENORMAL(X) has the value true if IEEE_SUPPORT_-
 14 DATATYPE(X) has the value true and the processor supports arithmetic operations and
 15 assignments with denormalized numbers (biased exponent $e = 0$ and fraction $f \neq 0$, see
 16 subclause 3.2 of the IEEE International Standard) for real variables of the same kind
 17 type parameter as X; otherwise, it has the value false.

18 *Case (ii):* IEEE_SUPPORT_DENORMAL() has the value true if and only if IEEE_SUPPORT_-
 19 DENORMAL(X) has the value true for all real X.

20 **Example.** IEEE_SUPPORT_DENORMAL(X) has the value true if the processor supports denormalized
 21 numbers for X.

NOTE 14.12

The denormalized numbers are not included in the 13.4 model for real numbers; they satisfy the inequality $\text{ABS}(X) < \text{TINY}(X)$. They usually occur as a result of an arithmetic operation whose exact result is less than $\text{TINY}(X)$. Such an operation causes IEEE_UNDERFLOW to signal unless the result is exact. IEEE_SUPPORT_DENORMAL(X) is false if the processor never returns a denormalized number as the result of an arithmetic operation.

22 **14.10.25 IEEE_SUPPORT_DIVIDE () or IEEE_SUPPORT_DIVIDE (X)**

23 **Description.** Inquire whether the processor supports divide with the accuracy specified by the IEEE
 24 International Standard.

25 **Class.** Inquiry function.

26 **Argument.** X shall be of type real. It may be a scalar or an array.

27 **Result Characteristics.** Default logical scalar.

28 **Result Value.**

29 *Case (i):* IEEE_SUPPORT_DIVIDE(X) has the value true if the processor supports divide with the
 30 accuracy specified by the IEEE International Standard for real variables of the same kind
 31 type parameter as X; otherwise, it has the value false.

32 *Case (ii):* IEEE_SUPPORT_DIVIDE() has the value true if and only if IEEE_SUPPORT_-
 33 DIVIDE(X) has the value true for all real X.

1 **Example.** IEEE_SUPPORT_DIVIDE(X) has the value true if the processor supports IEEE divide for
2 X.

3 **14.10.26 IEEE_SUPPORT_FLAG (FLAG) or IEEE_SUPPORT_FLAG (FLAG, X)**

4 **Description.** Inquire whether the processor supports an exception.

5 **Class.** Inquiry function.

6 **Arguments.**

FLAG shall be scalar and of type TYPE(IEEE_FLAG_TYPE). Its value shall be one
of IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDER-
FLOW, or IEEE_INEXACT.

7
8 X shall be of type real. It may be a scalar or an array.

9 **Result Characteristics.** Default logical scalar.

10 **Result Value.**

11 *Case (i):* IEEE_SUPPORT_FLAG(FLAG, X) has the value true if the processor supports detec-
12 tion of the specified exception for real variables of the same kind type parameter as X;
13 otherwise, it has the value false.

14 *Case (ii):* IEEE_SUPPORT_FLAG(FLAG) has the value true if and only if IEEE_SUPPORT_-
15 FLAG(FLAG, X) has the value true for all real X.

16 **Example.** IEEE_SUPPORT_FLAG(IEEE_INEXACT) has the value true if the processor supports the
17 inexact exception.

18 **14.10.27 IEEE_SUPPORT_HALTING (FLAG)**

19 **Description.** Inquire whether the processor supports the ability to control during program execution
20 whether to abort or continue execution after an exception.

21 **Class.** Inquiry function.

22 **Argument.** FLAG shall be scalar and of type TYPE(IEEE_FLAG_TYPE). Its value shall be one
23 of IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_-
24 INEXACT.

25 **Result Characteristics.** Default logical scalar.

26 **Result Value.** The result has the value true if the processor supports the ability to control during pro-
27 gram execution whether to abort or continue execution after the exception specified by FLAG; otherwise,
28 it has the value false. Support includes the ability to change the mode by CALL IEEE_SET_HALT-
29 ING(FLAG).

30 **Example.** IEEE_SUPPORT_HALTING(IEEE_OVERFLOW) has the value true if the processor sup-
31 ports control of halting after an overflow.

32 **14.10.28 IEEE_SUPPORT_INF () or IEEE_SUPPORT_INF (X)**

33 **Description.** Inquire whether the processor supports the IEEE infinity facility.

34 **Class.** Inquiry function.

35 **Argument.** X shall be of type real. It may be a scalar or an array.

1 **Result Characteristics.** Default logical scalar.

2 **Result Value.**

3 *Case (i):* IEEE_SUPPORT_INF(X) has the value true if the processor supports IEEE infinities
4 (positive and negative) for real variables of the same kind type parameter as X; otherwise,
5 it has the value false.

6 *Case (ii):* IEEE_SUPPORT_INF() has the value true if and only if IEEE_SUPPORT_INF(X) has
7 the value true for all real X.

8 **Example.** IEEE_SUPPORT_INF(X) has the value true if the processor supports IEEE infinities for X.

9 **14.10.29 IEEE_SUPPORT_IO () or IEEE_SUPPORT_IO (X)**

10 **Description.** Inquire whether the processor supports IEEE base conversion rounding during formatted
11 input/output (9.4.5.14, 9.5.2.13, 10.7.2.3.7).

12 **Class.** Inquiry function.

13 **Argument.** X shall be of type real. It may be a scalar or an array.

14 **Result Characteristics.** Default logical scalar.

15 **Result Value.**

16 *Case (i):* IEEE_SUPPORT_IO(X) has the value true if the processor supports IEEE base conversion
17 during formatted input/output (9.4.5.14, 9.5.2.13, 10.7.2.3.7) as described in the IEEE In-
18 ternational Standard for the modes UP, DOWN, ZERO, and NEAREST for real variables
19 of the same kind type parameter as X; otherwise, it has the value false.

20 *Case (ii):* IEEE_SUPPORT_IO() has the value true if and only if IEEE_SUPPORT_IO(X) has the
21 value true for all real X.

22 **Example.** IEEE_SUPPORT_IO(X) has the value true if the processor supports IEEE base conversion
23 for X.

24 **14.10.30 IEEE_SUPPORT_NAN () or IEEE_SUPPORT_NAN (X)**

25 **Description.** Inquire whether the processor supports the IEEE Not-a-Number facility.

26 **Class.** Inquiry function.

27 **Argument.** X shall be of type real. It may be a scalar or an array.

28 **Result Characteristics.** Default logical scalar.

29 **Result Value.**

30 *Case (i):* IEEE_SUPPORT_NAN(X) has the value true if the processor supports IEEE NaNs for
31 real variables of the same kind type parameter as X; otherwise, it has the value false.

32 *Case (ii):* IEEE_SUPPORT_NAN() has the value true if and only if IEEE_SUPPORT_NAN(X) has
33 the value true for all real X.

34 **Example.** IEEE_SUPPORT_NAN(X) has the value true if the processor supports IEEE NaNs for X.

35 **14.10.31 IEEE_SUPPORT_ROUNDING (ROUND_VALUE) or IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)**

Description. Inquire whether the processor supports a particular IEEE rounding mode.

1

2 **Class.** Inquiry function.3 **Arguments.**

4 ROUND_VALUE shall be of type TYPE(IEEE_ROUND_TYPE).

5 X shall be of type real. It may be a scalar or an array.

6 **Result Characteristics.** Default logical scalar.7 **Result Value.**

8 *Case (i):* IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X) has the value true if the processor
 9 supports the rounding mode defined by ROUND_VALUE for real variables of the same
 10 kind type parameter as X; otherwise, it has the value false. Support includes the ability
 11 to change the mode by CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE).

12 *Case (ii):* IEEE_SUPPORT_ROUNDING(ROUND_VALUE) has the value true if and only if IEEE-
 13 _SUPPORT_ROUNDING(ROUND_VALUE, X) has the value true for all real X.

14 **Example.** IEEE_SUPPORT_ROUNDING(IEEE_TO_ZERO) has the value true if the processor sup-
 15 ports rounding to zero for all reals.

16 **14.10.32 IEEE_SUPPORT_SQRT () or IEEE_SUPPORT_SQRT (X)**

17 **Description.** Inquire whether the intrinsic function SQRT conforms to the IEEE International Stan-
 18 dard.

19 **Class.** Inquiry function.20 **Argument.** X shall be of type real. It may be a scalar or an array.21 **Result Characteristics.** Default logical scalar.22 **Result Value.**

23 *Case (i):* IEEE_SUPPORT_SQRT(X) has the value true if the intrinsic function SQRT conforms
 24 to the IEEE International Standard for real variables of the same kind type parameter as
 25 X; otherwise, it has the value false.

26 *Case (ii):* IEEE_SUPPORT_SQRT() has the value true if and only if IEEE_SUPPORT_SQRT(X)
 27 has the value true for all real X.

28 **Example.** If IEEE_SUPPORT_SQRT(1.0) has the value true, SQRT(−0.0) will have the value −0.0.

29 **14.10.33 IEEE_SUPPORT_STANDARD () or IEEE_SUPPORT_STANDARD (X)**

30 **Description.** Inquire whether the processor supports all the IEEE facilities defined in this part of
 31 ISO/IEC 1539.

32 **Class.** Inquiry function.33 **Argument.** X shall be of type real. It may be a scalar or an array.34 **Result Characteristics.** Default logical scalar.35 **Result Value.**

36 *Case (i):* IEEE_SUPPORT_STANDARD(X) has the value true if the results of all the func-
 37 tions IEEE_SUPPORT_DATATYPE(X), IEEE_SUPPORT_DENORMAL(X), IEEE-

1 SUPPORT_DIVIDE(X), IEEE_SUPPORT_FLAG(FLAG,X) for valid FLAG, IEEE-
 2 __SUPPORT_HALTING(FLAG) for valid FLAG, IEEE_SUPPORT_INF(X), IEEE-
 3 __SUPPORT_NAN(X), IEEE_SUPPORT_ROUNDING (ROUND_VALUE,X) for valid
 4 ROUND_VALUE, and IEEE_SUPPORT_SQRT (X) are all true; otherwise, the result
 5 has the value false.

6 *Case (ii):* IEEE_SUPPORT_STANDARD() has the value true if and only if IEEE_SUPPORT_-
 7 STANDARD(X) has the value true for all real X.

8 **Example.** IEEE_SUPPORT_STANDARD() has the value false if the processor supports both IEEE
 9 and non-IEEE kinds of reals.

10 **14.10.34 IEEE_SUPPORT_UNDERFLOW_CONTROL() or** 11 **IEEE_SUPPORT_UNDERFLOW_CONTROL(X)**

12 **Description.** Inquire whether the procedure supports the ability to control the underflow mode during
 13 program execution.

14 **Class.** Inquiry function.

15 **Argument.** X shall be of type real. It may be a scalar or an array.

16 **Result Characteristics.** Default logical scalar.

17 **Result Value.**

18 *Case (i):* IEEE_SUPPORT_UNDERFLOW_CONTROL(X) has the value true if the processor sup-
 19 ports control of the underflow mode for floating-point calculations with the same type as
 20 X, and false otherwise.

21 *Case (ii):* IEEE_SUPPORT_UNDERFLOW_CONTROL() has the value true if the processor sup-
 22 ports control of the underflow mode for all floating-point calculations, and false otherwise.

23 **Example.** IEEE_SUPPORT_UNDERFLOW_CONTROL(2.5) has the value true if the processor sup-
 24 ports underflow mode control for calculations of type default real.

25 **14.10.35 IEEE_UNORDERED (X, Y)**

26 **Description.** IEEE unordered function. True if X or Y is a NaN, and false otherwise.

27 **Class.** Elemental function.

28 **Arguments.** The arguments shall be of type real.

29 **Restriction.** IEEE_UNORDERED(X,Y) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) or
 30 IEEE_SUPPORT_DATATYPE(Y) has the value false.

31 **Result Characteristics.** Default logical.

32 **Result Value.** The result has the value true if X or Y is a NaN or both are NaNs; otherwise, it has
 33 the value false.

34 **Example.** IEEE_UNORDERED(0.0,SQRT(-1.0)) has the value true if IEEE_SUPPORT_SQRT(1.0)
 35 has the value true.

36 **14.10.36 IEEE_VALUE (X, CLASS)**

37 **Description.** Generate an IEEE value.

1 **Class.** Elemental function.

2 **Arguments.**

3 X shall be of type real.

CLASS shall be of type 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_DENORMAL or IEEE_POSITIVE_-
DENORMAL if IEEE_SUPPORT_DENORMAL(X) has the value true, IEEE_NEGA-
TIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO or IEEE_-
POSITIVE_NORMAL.

5 **Restriction.** IEEE_VALUE(X,CLASS) shall not be invoked if IEEE_SUPPORT_DATATYPE(X) has
6 the value false.

7 **Result Characteristics.** Same as X.

8 **Result Value.** The result value is an IEEE value as specified by CLASS. Although in most cases the
9 value is processor dependent, the value shall not vary between invocations for any particular X kind
10 type parameter and CLASS value.

11 **Example.** IEEE_VALUE(1.0,IEEE_NEGATIVE_INF) has the value -infinity.

12 Whenever IEEE_VALUE returns a signaling NaN, it is processor dependent whether or not invalid is
13 raised and processor dependent whether or not the signaling NaN is converted into a quiet NaN.

NOTE 14.13

If the *expr* in an assignment statement is a reference to the IEEE_VALUE function that returns a signaling NaN and the *variable* is of the same type and kind as the function result, it is recommended that the signaling NaN be preserved.

14.11 Examples

NOTE 14.14

```
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
```


NOTE 14.14 (cont.)

```

        MATRIX_ERROR = .TRUE.
        RETURN
    END IF
! The processor ensures that IEEE_OVERFLOW is quiet
    MULT = 0.0
    DO I = 1, SIZE(A)
        MULT = MULT + A(I)*B(I)
    END DO
    CALL IEEE_GET_FLAG(IEEE_OVERFLOW,OVERFLOW)
    IF (OVERFLOW) MATRIX_ERROR = .TRUE.
END FUNCTION MULT
END MODULE DOT

```

This module provides the dot product of two real arrays of rank 1. If the sizes of the arrays are different, an immediate return occurs with MATRIX_ERROR true. If overflow occurs during the actual calculation, the IEEE_OVERFLOW flag will signal and MATRIX_ERROR will be true.

NOTE 14.15

```

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
END IF
CALL IEEE_SET_STATUS(STATUS_VALUE)

```

NOTE 14.15 (cont.)

In this example, the function FAST_INV might cause a condition to signal. If it does, another try is made with SLOW_INV. If this still fails, a message is printed and the program stops. Note, also, that it is important to set the flags quiet before the second try. The state of all the flags is stored and restored.

NOTE 14.16

```

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.

NOTE 14.17

```

REAL FUNCTION HYPOT(X, Y)
! In rare circumstances this may lead to the signaling of IEEE_OVERFLOW
! The caller needs to ensure that exceptions do not cause halting.
USE, INTRINSIC :: IEEE_ARITHMETIC
USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_UNDERFLOW_FLAG
! IEEE_OVERFLOW is always available with IEEE_ARITHMETIC
REAL X, Y
REAL SCALED_X, SCALED_Y, SCALED_RESULT
LOGICAL, DIMENSION(2) :: FLAGS
TYPE(IEEE_FLAG_TYPE), PARAMETER, DIMENSION(2) :: &
    OUT_OF_RANGE = (/ IEEE_OVERFLOW, IEEE_UNDERFLOW /)
INTRINSIC SQRT, ABS, EXPONENT, MAX, DIGITS, SCALE
! The processor clears the flags on entry
! Try a fast algorithm first
HYPOT = SQRT( X**2 + Y**2 )
CALL IEEE_GET_FLAG(OUT_OF_RANGE,FLAGS)

```

NOTE 14.17 (cont.)

```

IF ( ANY(FLAGS) ) THEN
  CALL IEEE_SET_FLAG(OUT_OF_RANGE,.FALSE.)
  IF ( X==0.0 .OR. Y==0.0 ) THEN
    HYPOT = ABS(X) + ABS(Y)
  ELSE IF ( 2*ABS(EXPONENT(X)-EXPONENT(Y)) > DIGITS(X)+1 ) THEN
    HYPOT = MAX( ABS(X), ABS(Y) )! one of X and Y can be ignored
  ELSE ! scale so that ABS(X) is near 1
    SCALED_X = SCALE( X, -EXPONENT(X) )
    SCALED_Y = SCALE( Y, -EXPONENT(X) )
    SCALED_RESULT = SQRT( SCALED_X**2 + SCALED_Y**2 )
    HYPOT = SCALE( SCALED_RESULT, EXPONENT(X) ) ! might cause overflow
  END IF
END IF
! The processor resets any flag that was signaling on entry
END FUNCTION HYPOT

```

An attempt is made to evaluate this function directly in the fastest possible way. This will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation might involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if X and Y are both near the overflow limit). If the IEEE_OVERFLOW or IEEE_UNDERFLOW flag is signaling on entry, it is reset on return by the processor, so that earlier exceptions are not lost.

15 Interoperability with C

15.1 General

Fortran provides a means of referencing procedures that are defined by means of the C programming language or procedures that can be described by C prototypes as defined in 6.7.5.3 of the C International Standard, 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 that have external linkage as defined in 6.2.2 of the C International Standard.

The ISO_C_BINDING module provides access to named constants that represent kind type parameters of data representations compatible with C types. Fortran also provides facilities for defining derived types (4.5) and enumerations (4.6) that correspond to C types.

15.2 The ISO_C_BINDING intrinsic module

15.2.1 Summary of contents

The processor shall provide the intrinsic module ISO_C_BINDING. This module shall make accessible the following entities: named constants with the names listed in the second column of Table 15.2 and the first column of Table 15.1, the procedures specified in 15.2.3, C_PTR, C_FUNPTR, C_NULL_PTR, and C_NULL_FUNPTR. A processor may provide other public entities in the ISO_C_BINDING intrinsic module in addition to those listed here.

NOTE 15.1

To avoid potential name conflicts with program entities, it is recommended that a program use the ONLY option in any USE statement that references the ISO_C_BINDING intrinsic module.

15.2.2 Named constants and derived types in the module

The entities listed in the second column of Table 15.2, shall be named constants of type default integer.

The value of C_INT shall be a valid value for an integer kind parameter on the processor. The values of C_SHORT, C_LONG, C_LONG_LONG, C_SIGNED_CHAR, C_SIZE_T, C_INT8_T, C_INT16_T, C_INT32_T, C_INT64_T, C_INT_LEAST8_T, C_INT_LEAST16_T, C_INT_LEAST32_T, C_INT_LEAST64_T, C_INT_FAST8_T, C_INT_FAST16_T, C_INT_FAST32_T, C_INT_FAST64_T, C_INTMAX_T, and C_INT_PTR_T shall each be a valid value for an integer kind type parameter on the processor or shall be -1 if the companion C processor defines the corresponding C type and there is no interoperating Fortran processor kind or -2 if the C processor does not define the corresponding C type.

The values of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE shall each be a valid value for a real kind type parameter on the processor or shall be -1 if the C processor's type does not have a precision equal to the precision of any of the Fortran processor's real kinds, -2 if the C processor's type does not have a range equal to the range of any of the Fortran processor's real kinds, -3 if the C processor's type has neither the precision nor range of any of the Fortran processor's real kinds, and equal to -4 if there is no interoperating Fortran processor kind for other reasons. The values of C_FLOAT_COMPLEX, C_DOUBLE_COMPLEX, and C_LONG_DOUBLE_COMPLEX shall be the same as those of C_FLOAT,

1 C_DOUBLE, and C_LONG_DOUBLE, respectively.

NOTE 15.2

If the C processor supports more than one variety of float, double or long double, the Fortran processor might find it helpful to select from among more than one ISO_C_BINDING module by a processor dependent means.

2 The value of C_BOOL shall be a valid value for a logical kind parameter on the processor or shall be -1.

3 The value of C_INT_BITS shall be a valid value for a bits kind type parameter of the processor.

4 The values of C_SHORT_BITS, C_LONG_BITS, C_LONG_LONG_BITS, C_SIGNED_CHAR_BITS, C_-
5 INT8_T_BITS, C_INT16_T_BITS, C_INT32_T_BITS, C_INT64_T_BITS, C_INT_LEAST8_T_BITS, C_-
6 INT_LEAST16_T_BITS, C_INT_LEAST32_T_BITS, C_INT_LEAST64_T_BITS, C_INT_FAST8_T_BITS,
7 C_INT_FAST16_T_BITS, C_INT_FAST32_T_BITS, C_INT_FAST64_T_BITS, C_INTMAX_T_BITS, C_-
8 INTPTR_T_BITS, and C_BOOL_BITS shall each be a valid value for a bits kind type parameter on the
9 processor, -1 if the companion C processor defines the corresponding C type and there is no interoper-
10 ating Fortran processor kind, or -2 if the C processor does not define the corresponding C type.

11 The value of C_CHAR shall be a valid value for a character kind type parameter on the processor or
12 shall be -1. The value of C_CHAR is known as the **C character kind**.

13 The following entities shall be named constants of type character with a length parameter of one. The
14 kind parameter value shall be equal to the value of C_CHAR unless C_CHAR = -1, in which case the
15 kind parameter value shall be the same as for default kind. The values of these constants are specified
16 in Table 15.1. In the case that C_CHAR $\neq$ -1 the value is specified using C syntax. The semantics of
17 these values are explained in 5.2.1 and 5.2.2 of the C International Standard.

Table 15.1: 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'

NOTE 15.3

The value of NEW_LINE(C_NEW_LINE) is C_NEW_LINE (13.7.127).

18 The entities C_PTR and C_FUNPTR are described in 15.3.3.

19 The entity C_NULL_PTR shall be a named constant of type C_PTR. The value of C_NULL_PTR shall
20 be the same as the value NULL in C. The entity C_NULL_FUNPTR shall be a named constant of type
21 C_FUNPTR. The value of C_NULL_FUNPTR shall be that of a null pointer to a function in C.

22 15.2.3 Procedures in the module

23 In the detailed descriptions below, procedure names are generic and not specific.

24 A C procedure argument is often defined in terms of a **C address**. The C_LOC and C_FUNLOC

1 functions are provided so that Fortran applications can determine the appropriate value to use with C
 2 facilities. The C_ASSOCIATED function is provided so that Fortran programs can compare C addresses.
 3 The C_F_POINTER and C_F_PROCPOINTER subroutines provide a means of associating a Fortran
 4 pointer with the target of a C pointer.

5 15.2.3.1 C_ASSOCIATED (C_PTR_1 [, C_PTR_2])

6 **Description.** Indicates the association status of C_PTR_1 or indicates whether C_PTR_1 and C_PTR_2
 7 are associated with the same entity.

8 **Class.** Inquiry function.

9 **Arguments.**

10 C_PTR_1 shall be a scalar of type C_PTR or C_FUNPTR.

C_PTR_2 shall be a scalar of the same type as C_PTR_1.
 11 (optional)

12 **Result Characteristics.** Default logical scalar.

13 **Result Value.**

14 *Case (i):* If C_PTR_2 is absent, the result is false if C_PTR_1 is a C null pointer and true otherwise.

15 *Case (ii):* If C_PTR_2 is present, the result is false if C_PTR_1 is a C null pointer. Otherwise, the
 16 result is true if C_PTR_1 compares equal to C_PTR_2 in the sense of 6.3.2.3 and 6.5.9 of
 17 the C International Standard, and false otherwise.

NOTE 15.4

The following example illustrates the use of C_LOC and C_ASSOCIATED.

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_PTR, C_FLOAT, C_ASSOCIATED, C_LOC
INTERFACE
  SUBROUTINE FOO(GAMMA) BIND(C)
    IMPORT C_PTR
    TYPE(C_PTR), VALUE :: GAMMA
  END SUBROUTINE FOO
END INTERFACE
REAL(C_FLOAT), TARGET, DIMENSION(100) :: ALPHA
TYPE(C_PTR) :: BETA
...
IF (.NOT. C_ASSOCIATED(BETA)) THEN
  BETA = C_LOC(ALPHA)
ENDIF
CALL FOO(BETA)
```

18 15.2.3.2 C_F_POINTER (CPTR, FPTR [, SHAPE])

19 **Description.** Associates a data pointer with the target of a C pointer and specifies its shape.

20 **Class.** Subroutine.

21 **Arguments.**

CPTR shall be a scalar of type C_PTR. It is an INTENT(IN) argument. Its value shall be

- (1) the C address of an interoperable data entity, or
- (2) the result of a reference to C_LOC with a noninteroperable argument.

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, and shall not be a co-indexed object. It is an INTENT(OUT) argument.

If the value of CPTR is the C address of an interoperable data entity, FPTR shall be a data pointer with type and type parameters interoperable with the type of the entity. In this case, FPTR becomes pointer associated with the target of CPTR. If FPTR is an array, its shape is specified by SHAPE and each lower bound is 1.

If the value of CPTR is the result of a reference to C_LOC with a noninteroperable argument X, FPTR shall be a nonpolymorphic scalar pointer with the same type and type parameters as X. In this case, X or its target if it is a pointer shall not have been deallocated or have become undefined due to execution of a RETURN or END statement since the reference. FPTR becomes pointer associated with X or its target.

SHAPE shall be of type integer and rank one. It is an INTENT(IN) argument. SHAPE shall (optional) be present if and only if FPTR is an array; its size shall be equal to the rank of FPTR.

15.2.3.3 C_F_PROCPOINTER (CPTR, FPTR)

Description. Associates a procedure pointer with the target of a C function pointer.

Class. Subroutine.

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.

FPTR shall be a procedure pointer, and shall not be a co-indexed object. It is an INTENT(OUT) argument. The interface for FPTR shall be interoperable with the prototype that describes the target of CPTR. FPTR becomes pointer associated with the target of CPTR.

NOTE 15.5

The term “target” in the descriptions of C_F_POINTER and C_F_PROCPOINTER denotes the entity referenced by a C pointer, as described in 6.2.5 of the C International Standard.

15.2.3.4 C_FUNLOC (X)

Description. Returns the C address of the argument.

Class. Inquiry function.

Argument. X shall either be a procedure that is interoperable, or a procedure pointer associated with an interoperable procedure. It shall not be a co-indexed object.

Result Characteristics. Scalar of type C_FUNPTR.

Result Value.

The result value is described using the result name CPTR. The result is determined as if C_FUNPTR were a derived type containing an implicit-interface procedure pointer component PX

1 and the pointer assignment CPTR%PX => X were executed.

2 The result is a value that can be used as an actual CPTR argument in a call to C_F_PROC-
3 POINTER where FPTR has attributes that would allow the pointer assignment FPTR => X.
4 Such a call to C_F_PROCPTR shall have the effect of the pointer assignment FPTR =>
5 X.

6 15.2.3.5 C_LOC (X)

7 **Description.** Returns the C address of the argument.

8 **Class.** Inquiry function.

9 **Argument.** X shall have either the POINTER or TARGET attribute. It shall not be a co-indexed
10 object. It shall either be a contiguous variable with interoperable type and type parameters, or be a
11 scalar, nonpolymorphic variable with no length type parameters. If it is allocatable, it shall be allocated.
12 If it is a pointer, it shall be associated. If it is an array, it shall have nonzero size.

13 **Result Characteristics.** Scalar of type C_PTR.

14 **Result Value.**

15 The result value is described using the result name CPTR.

16 If X is a scalar data entity, the result is determined as if C_PTR were a derived type containing a scalar
17 pointer component PX of the type and type parameters of X and the pointer assignment CPTR%PX
18 => X were executed.

19 If X is an array data entity, the result is determined as if C_PTR were a derived type containing a scalar
20 pointer component PX of the type and type parameters of X and the pointer assignment of CPTR%PX
21 to the first element of X were executed.

22 If X is a data entity that is interoperable or has interoperable type and type parameters, the
23 result is the value that the C processor returns as the result of applying the unary "&" operator
24 (as defined in the C International Standard, 6.5.3.2) to the target of CPTR%PX.

25 The result is a value that can be used as an actual CPTR argument in a call to C_F_POINTER
26 where FPTR has attributes that would allow the pointer assignment FPTR => X. Such a call
27 to C_F_POINTER shall have the effect of the pointer assignment FPTR => X.

NOTE 15.6

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, portable C functions should treat it as a void (generic) C pointer that cannot be dereferenced (6.5.3.2 in the C International Standard).

28 15.2.3.6 C_SIZEOF (X)

29 **Description.** Returns the size of X in bytes.

30 **Class.** Inquiry function.

31 **Argument.** X shall be an interoperable data entity that is not an assumed-size array.

32 **Result Characteristics.** Scalar integer of kind C_SIZE_T (15.3.2).

33 **Result Value.**

1 If X is scalar, the result value is the value that the C processor returns as the result of applying
 2 the sizeof operator (C International Standard, subclause 6.5.3.4) to an object of a type that
 3 interoperates with the type and type parameters of X.

4 If X is an array, the result value is the value that the C processor returns as the result of applying
 5 the sizeof operator to an object of a type that interoperates with the type and type parameters of X,
 6 multiplied by the number of elements in X.

7 15.3 Interoperability between Fortran and C entities

8 15.3.1 General

9 Subclause 15.3 defines the conditions under which a Fortran entity is interoperable. If a Fortran entity is
 10 interoperable, an equivalent entity may be defined by means of C and the Fortran entity **interoperates**
 11 with the C entity. There does not have to be such an interoperating C entity.

NOTE 15.7

A Fortran entity can be interoperable with more than one C entity.

12 15.3.2 Interoperability of intrinsic types

13 Table 15.2 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic
 14 type with particular type parameter values is interoperable with a C type if the type and kind type
 15 parameter value are listed in the table on the same row as that C type; if the type is character, inter-
 16 operability also requires that the length type parameter be omitted or be specified by an initialization
 17 expression whose value is one. A combination of Fortran type and type parameters that is interoperable
 18 with a C type listed in the table is also interoperable with any unqualified C type that is compatible
 19 with the listed C type.

20 The second column of the table refers to the named constants made accessible by the ISO_C_BINDING
 21 intrinsic module. If the value of any of these named constants is negative, there is no combination of
 22 Fortran type and type parameters interoperable with the C type shown in that row.

23 A combination of intrinsic type and type parameters is **interoperable** if it is interoperable with a C
 24 type.

Table 15.2: Interoperability between Fortran and C types

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
	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

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
INTEGER	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
REAL	C_FLOAT	float
	C_DOUBLE	double
	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
BITS	C_INT_BITS	unsigned int or int
	C_SHORT_BITS	unsigned short or short
	C_LONG_BITS	unsigned long or long
	C_LONG_LONG_BITS	unsigned long long long long
	C_SIGNED_CHAR_BITS	unsigned char signed char
	C_INT8_T_BITS	uint8_t or int8_t
	C_INT16_T_BITS	uint16_t or int16_t
	C_INT32_T_BITS	uint32_t or int32_t
	C_INT64_T_BITS	uint64_t or int64_t
	C_INT_LEAST8_T_BITS	uint_least8_t int_least8_t
	C_INT_LEAST16_T_BITS	uint_least16_t int_least16_t
	C_INT_LEAST32_T_BITS	uint_least32_t int_least32_t
	C_INT_LEAST64_T_BITS	uint_least64_t int_least64_t
	C_INT_FAST8_T_BITS	uint_fast8_t or int_fast8_t
	C_INT_FAST16_T_BITS	uint_fast16_t int_fast16_t
	C_INT_FAST32_T_BITS	uint_fast32_t int_fast32_t

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_INT_FAST64_T_BITS	uint_fast64_t int_fast64_t
	C_INTMAX_T_BITS	uintmax_t or intmax_t
	C_INTPTR_T_BITS	uintptr_t or intptr_t
	C_BOOL_BITS	_Bool
CHARACTER	C_CHAR	char
The above mentioned C types are defined in the C International Standard, subclauses 6.2.5, 7.17, and 7.18.1.		

NOTE 15.8

For example, the type integer with a kind type parameter of C_SHORT is interoperable with the C type short or any C type derived (via typedef) from short.

NOTE 15.9

The C International Standard specifies that the representations for nonnegative signed integers are the same as the corresponding values of unsigned integers. 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.

NOTE 15.10

If a variable of type bits is the actual argument corresponding to an unsigned integer parameter of a C function and is interoperable with that parameter, or the unsigned integer result of a C function is assigned to a variable of type bits that is interoperable with the function result, the I format can be used to output the correct form of the unsigned integer value.

1 15.3.3 Interoperability with C pointer types

- 2 C_PTR and C_FUNPTR shall be derived types with only private components. No direct component of
- 3 either of these types is allocatable or a pointer. C_PTR is interoperable with any C object pointer type.
- 4 C_FUNPTR is interoperable with any C function pointer type.

NOTE 15.11

This implies that a C processor is required to have the same representation method for all C object pointer types and the same representation method for all C function pointer types if the C processor is to be the target of interoperability of a Fortran processor. The C International Standard does not impose this requirement.

NOTE 15.12

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 may 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.

NOTE 15.12 (cont.)

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.

1 15.3.4 Interoperability of derived types and C struct types

2 A Fortran derived type is **interoperable** if it has the BIND attribute.

3 C1501 (R430) A derived type with the BIND attribute shall not be a SEQUENCE type.

4 C1502 (R430) A derived type with the BIND attribute shall not have type parameters.

5 C1503 (R430) A derived type with the BIND attribute shall not have the EXTENDS attribute.

6 C1504 (R430) A derived type with the BIND attribute shall not have a *type-bound-procedure-part*.

7 C1505 (R430) Each component of a derived type with the BIND attribute shall be a nonpointer,
8 nonallocatable data component with interoperable type and type parameters.

NOTE 15.13

The syntax rules and their constraints require that a derived type that is interoperable have components that are all data objects that are interoperable. No component is permitted to be a procedure or allocatable, but a component of type C_FUNPTR or C_PTR may hold the C address of such an entity.

9 A Fortran derived type is interoperable with a C struct type if the derived-type definition of the Fortran
10 type specifies BIND(C) (4.5.2), the Fortran derived type and the C struct type have the same number
11 of components, and the components of the Fortran derived type have types and type parameters that
12 are interoperable with the types of the corresponding components of the struct type. A component of
13 a Fortran derived type and a component of a C struct type correspond if they are declared in the same
14 relative position in their respective type definitions.

NOTE 15.14

The names of the corresponding components of the derived type and the C struct type need not be the same.

15 There is no Fortran type that is interoperable with a C struct type that contains a bit field or that
16 contains a flexible array member. There is no Fortran type that is interoperable with a C union type.

NOTE 15.15

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
```

NOTE 15.15 (cont.)

```

    INTEGER(C_INT) :: I, J
    REAL(C_FLOAT) :: S
END TYPE MYFTYPE

```

The names of the types and the names of the components are not significant for the purposes of determining whether a Fortran derived type is interoperable with a C struct type.

NOTE 15.16

The C International Standard requires the names and component names to be the same in order for the types to be compatible (C International Standard, subclause 6.2.7). This is similar to Fortran's rule describing when different derived type definitions describe the same sequence type. This rule was not extended to determine whether a Fortran derived type is interoperable with a C struct type because the case of identifiers is significant in C but not in Fortran.

1 15.3.5 Interoperability of scalar variables

2 A scalar Fortran variable is **interoperable** if its type and type parameters are interoperable and it has
3 neither the pointer nor the allocatable attribute.

4 An interoperable scalar Fortran variable is interoperable with a scalar C entity if their types and type
5 parameters are interoperable.

6 15.3.6 Interoperability of array variables

7 An array Fortran variable is **interoperable** if its type and type parameters are interoperable and it is
8 of explicit shape or assumed size.

9 An explicit-shape or assumed-size array of rank r , with a shape of $[e_1 \dots e_r]$ is interoperable with
10 a C array if its size is nonzero and

11 (1) either

12 (a) the array is assumed size, and the C array does not specify a size, or

13 (b) the array is an explicit shape array, and the extent of the last dimension (e_r) is the
14 same as the size of the C array, and

15 (2) either

16 (a) r is equal to one, and an element of the array is interoperable with an element of the
17 C array, or

18 (b) r is greater than one, and an explicit-shape array with shape of $[e_1 \dots e_{r-1}]$,
19 with the same type and type parameters as the original array, is interoperable with a
20 C array of a type equal to the element type of the original C array.

NOTE 15.17

An element of a multi-dimensional C array is an array type, so a Fortran array of rank one is not interoperable with a multidimensional C array.

NOTE 15.18

An allocatable or pointer array is never interoperable. Such an array does not meet the requirement of being explicit shape or assumed size.

NOTE 15.19

For example, a Fortran array declared as

```
INTEGER :: A(18, 3:7, *)
```

is interoperable with a C array declared as

```
int b[][5][18]
```

NOTE 15.20

The C programming language defines null-terminated strings, which are actually arrays of the C type `char` that have a C null character in them to indicate the last valid element. A Fortran array of type character with a kind type parameter equal to `C_CHAR` is interoperable with a C string.

Fortran's rules of sequence association (12.5.2.12) permit a character scalar actual argument to be associated with a dummy argument array. This makes it possible to argument associate a Fortran character string with a C string.

Note 15.24 has an example of interoperation between Fortran and C strings.

1 15.3.7 Interoperability of procedures and procedure interfaces

2 A Fortran procedure is **interoperable** if it has the `BIND` attribute, that is, if its interface is specified
3 with a *proc-language-binding-spec*.

4 A Fortran procedure interface is interoperable with a C function prototype if

- 5 (1) the interface has the `BIND` attribute,
- 6 (2) either
 - 7 (a) the interface describes a function whose result variable is a scalar that is interoperable
 - 8 with the result of the prototype or
 - 9 (b) the interface describes a subroutine and the prototype has a result type of void,
- 10 (3) the number of dummy arguments of the interface is equal to the number of formal parameters
- 11 of the prototype,
- 12 (4) any dummy argument with the `VALUE` attribute is interoperable with the corresponding
- 13 formal parameter of the prototype,
- 14 (5) any dummy argument without the `VALUE` attribute corresponds to a formal parameter
- 15 of the prototype that is of a pointer type, and the dummy argument is interoperable with
- 16 an entity of the referenced type (C International Standard, 6.2.5, 7.17, and 7.18.1) of the
- 17 formal parameter, and
- 18 (6) the prototype does not have variable arguments as denoted by the ellipsis (...).

NOTE 15.21

The **referenced type** of a C pointer type is the C type of the object that the C pointer type points to. For example, the referenced type of the pointer type `int *` is `int`.

NOTE 15.22

The C language allows specification of a C function that can take a variable number of arguments (C International Standard, 7.15). This part of ISO/IEC 1539 does not provide a mechanism for Fortran procedures to interoperate with such C functions.

- 1 A formal parameter of a C function prototype corresponds to a dummy argument of a Fortran interface if
- 2 they are in the same relative positions in the C parameter list and the dummy argument list, respectively.

NOTE 15.23

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 may correspond to a Fortran dummy argument of type C_PTR with the VALUE attribute or to a Fortran scalar that does not have the VALUE attribute. In the above example, the C pointers j and k correspond to the Fortran scalars J and K, respectively, and the C pointer m corresponds to the Fortran dummy argument M of type C_PTR.

NOTE 15.24

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 may 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) :: &
```


NOTE 15.24 (cont.)

```

&      DIGIT_STRING = C_CHAR_'123456789' // C_NULL_CHAR
CHARACTER(KIND=C_CHAR) :: DIGIT_ARR(10)

CALL COPY(DIGIT_STRING, DIGIT_ARR)
PRINT '(1X, A1)', DIGIT_ARR(1:9)
END

```

The procedure reference has character string actual arguments. These correspond to character array dummy arguments in the procedure interface body as allowed by Fortran's rules of sequence association (12.5.2.12). 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.

1 15.4 Interoperation with C global variables

2 15.4.1 General

3 A C variable with external linkage may interoperate with a common block or with a variable declared
4 in the scope of a module. The common block or variable shall be specified to have the BIND attribute.

5 At most one variable that is associated with a particular C variable with external linkage is permitted
6 to be declared within a program. A variable shall not be initially defined by more than one processor.

7 If a common block is specified in a BIND statement, it shall be specified in a BIND statement with
8 the same binding label in each scoping unit in which it is declared. A C variable with external linkage
9 interoperates with a common block that has been specified in a BIND statement

10 (1) if the C variable is of a struct type and the variables that are members of the common block
11 are interoperable with corresponding components of the struct type, or

12 (2) if the common block contains a single variable, and the variable is interoperable with the C
13 variable.

14 There does not have to be an associated C entity for a Fortran entity with the BIND attribute.

NOTE 15.25

The following are examples of the usage of the BIND attribute for variables and for a common block. The Fortran variables, C_EXTERN and C2, interoperate with the C variables, c_extern and myVariable, respectively. The Fortran common blocks, COM and SINGLE, interoperate with the C variables, com and single, respectively.

```

MODULE LINK_TO_C_VARS
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(C_INT), BIND(C) :: C_EXTERN
  INTEGER(C_LONG) :: C2
  BIND(C, NAME='myVariable') :: C2

  COMMON /COM/ R, S
  REAL(C_FLOAT) :: R, S, T
  BIND(C) :: /COM/, /SINGLE/

```

NOTE 15.25 (cont.)

```

COMMON /SINGLE/ T
END MODULE LINK_TO_C_VARS
int c_extern;
long myVariable;
struct {float r, s;} com;
float single;

```

1 15.4.2 Binding labels for common blocks and variables

2 The **binding label** of a variable or common block is a value of type default character that specifies the
3 name by which the variable or common block is known to the companion processor.

4 If a variable or common block has the BIND attribute with the NAME= specifier and the value of its
5 expression, after discarding leading and trailing blanks, has nonzero length, the variable or common
6 block has this as its binding label. The case of letters in the binding label is significant. If a variable
7 or common block has the BIND attribute specified without a NAME= specifier, the binding label is the
8 same as the name of the entity using lower case letters. Otherwise, the variable or common block has
9 no binding label.

10 The binding label of a C variable with external linkage is the same as the name of the C variable. A
11 Fortran variable or common block with the BIND attribute that has the same binding label as a C
12 variable with external linkage is linkage associated (16.5.1.5) with that variable.

13 15.5 Interoperation with C functions

14 15.5.1 Definition and reference of interoperable procedures

15 A procedure that is interoperable may be defined either by means other than Fortran or by means of a
16 Fortran subprogram, but not both.

17 If the procedure is defined by means other than Fortran, it shall

- 18 (1) be describable by a C prototype that is interoperable with the interface,
- 19 (2) have external linkage as defined by 6.2.2 of the C International Standard, and
- 20 (3) have the same binding label as the interface.

21 A reference to such a procedure causes the function described by the C prototype to be called as specified
22 in the C International Standard.

23 A reference in C to a procedure that has the BIND attribute, has the same binding label, and is defined
24 by means of Fortran, causes the Fortran procedure to be invoked.

25 A procedure defined by means of Fortran shall not invoke setjmp or longjmp (C International Standard,
26 7.13). If a procedure defined by means other than Fortran invokes setjmp or longjmp, that procedure
27 shall not cause any procedure defined by means of Fortran to be invoked. A procedure defined by means
28 of Fortran shall not be invoked as a signal handler (C International Standard, 7.14.1).

29 If a procedure defined by means of Fortran and a procedure defined by means other than Fortran perform
30 input/output operations on the same external file, the results are processor dependent (9.4.3).

15.5.2 Binding labels for procedures

1

2 The **binding label** of a procedure is a value of type default character that specifies the name by which
 3 a procedure with the BIND attribute is known to the companion processor.

4 If a procedure has the BIND attribute with the NAME= specifier and the value of its expression, after
 5 discarding leading and trailing blanks, has nonzero length, the procedure has this as its binding label.
 6 The case of letters in the binding label is significant. If a procedure has the BIND attribute with
 7 no NAME= specifier, and the procedure is not a dummy procedure, internal procedure, or procedure
 8 pointer, then the binding label of the procedure is the same as the name of the procedure using lower
 9 case letters. Otherwise, the procedure has no binding label.

10 C1506 A procedure defined in a submodule shall not have a binding label unless its interface is declared
 11 in the ancestor module.

12 The binding label for a C function with external linkage is the same as the C function name.

NOTE 15.26

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
```

The C International Standard 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 may 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.

13 15.5.3 Exceptions and IEEE arithmetic procedures

14 A procedure defined by means other than Fortran shall not use signal (C International Standard, 7.14.1)
 15 to change the handling of any exception that is being handled by the Fortran processor.

16 A procedure defined by means other than Fortran shall not alter the floating-point status (14.6) other
 17 than by setting an exception flag to signaling.

18 The values of the floating-point exception flags on entry to a procedure defined by means other than
 19 Fortran are processor-dependent.

16 Scope, association, and definition

16.1 Identifiers and entities

Entities are identified by identifiers within a **scope** that is a program, a scoping unit, a construct, a single statement, or part of a statement.

- A **global identifier** has a scope of a program (2.2.2);
- A **local identifier** has a scope of a scoping unit (2.2);
- An identifier of a **construct entity** has a scope of a construct (7.2.3, 7.2.4, 8.1);
- An identifier of a **statement entity** has a scope of a statement or part of a statement (3.3).

An entity may be identified by

- (1) an image index (2.3.2),
- (2) a name (3.2.1),
- (3) a statement label (3.2.4),
- (4) an external input/output unit number (9.4),
- (5) an identifier of a pending data transfer operation (9.5.2.9, 9.6),
- (6) a submodule identifier (11.2.3),
- (7) a generic identifier (12.4.3.2), or
- (8) a binding label (15.5.2, 15.4.2).

By means of association, an entity may be referred to by the same identifier or a different identifier in a different scoping unit, or by a different identifier in the same scoping unit.

16.2 Scope of global identifiers

Program units, common blocks, external procedures, entities with binding labels, and images are **global entities** of a program. The name of a non-submodule program unit, common block, or external procedure is a global identifier and shall not be the same as the name of any other such global entity in the same program, except that an intrinsic module may have the same name as another program unit, common block, or external procedure in the same program. The submodule identifier of a submodule is a global identifier and shall not be the same as the submodule identifier of any other submodule. A binding label of an entity of the program is a global identifier and shall not be the same as the binding label of any other entity of the program; nor shall it be the same as the name of any other global entity of the program that is not an intrinsic module, ignoring differences in case. An entity of the program shall not be identified by more than one binding label.

NOTE 16.1

The name of a global entity may be the same as a binding label that identifies the same global entity.

NOTE 16.2

Of the various types of procedures, only external procedures have global names. An implementation may wish to assign global names to other entities in the Fortran program such as internal procedures, intrinsic procedures, procedures implementing intrinsic operators, procedures implementing input/output operations, etc. If this is done, it is the responsibility of the processor to ensure that none of these names conflicts with any of the names of the external procedures, with other globally named entities in a standard-conforming program, or with each other. For example, this might be done by including in each such added name a character that is not allowed in a standard-conforming name or by using such a character to combine a local designation with the global name of the program unit in which it appears.

NOTE 16.3

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.

- 1 External input/output units and pending data transfer operations are global entities.

2 **16.3 Scope of local identifiers**

3 **16.3.1 Classes of local identifiers**

4 Within a scoping unit, identifiers of entities in the classes

- 5 (1) named variables that are not statement or construct entities (16.4), named constants, named
- 6 constructs, statement functions, internal procedures, module procedures, dummy procedures,
- 7 intrinsic procedures, abstract interfaces, module procedure interfaces, generic interfaces, de-
- 8 rived types, namelist groups, external procedures accessed via USE, macros, and statement
- 9 labels,
- 10 (2) type parameters, components, and type-bound procedure bindings, in a separate class for
- 11 each type, and
- 12 (3) argument keywords, in a separate class for each procedure with an explicit interface

13 are local identifiers in that scoping unit.

14 Within a scoping unit, a local identifier of an entity of class (1) shall not be the same as a global identifier

15 used in that scoping unit unless the global identifier

- 16 (1) is used only as the *use-name* of a *rename* in a USE statement,
- 17 (2) is a common block name (16.3.2),
- 18 (3) is an external procedure name that is also a generic name, or
- 19 (4) is an external function name and the scoping unit is its defining subprogram (16.3.3).

20 Within a scoping unit, a local identifier of one class shall not be the same as another local identifier of

21 the same class, except that a generic name may be the same as the name of a procedure as explained

22 in 12.4.3.2 or the same as the name of a derived type (4.5.10), and a separate module procedure shall

23 have the same name as its corresponding module procedure interface body (12.6.2.5). A local identifier

24 of one class may be the same as a local identifier of another class.

NOTE 16.4

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

NOTE 16.4 (cont.)

```
SUBROUTINE SUB
```

```
...
```

```
A = SIN (K)
```

```
...
```

```
CONTAINS
```

```
FUNCTION SIN (X)
```

```
...
```

```
END FUNCTION SIN
```

```
END SUBROUTINE SUB
```

any reference to function SIN in subroutine SUB refers to the internal function SIN, not to the intrinsic function of the same name.

1 A local identifier identifies an entity in a scoping unit and may be used to identify an entity in another
2 scoping unit except in the following cases.

- 3 (1) The name that appears as a *subroutine-name* in a *subroutine-stmt* has limited use within
4 the scope established by the *subroutine-stmt*. It can be used to identify recursive references
5 of the subroutine or to identify a common block (the latter is possible only for internal and
6 module subroutines).
- 7 (2) The name that appears as a *function-name* in a *function-stmt* has limited use within the
8 scope established by that *function-stmt*. It can be used to identify the result variable, to
9 identify recursive references of the function, or to identify a common block (the latter is
10 possible only for internal and module functions).
- 11 (3) The name that appears as an *entry-name* in an *entry-stmt* has limited use within the scope
12 of the subprogram in which the *entry-stmt* appears. It can be used to identify the result
13 variable if the subprogram is a function, to identify recursive references, or to identify a
14 common block (the latter is possible only if the *entry-stmt* is in a module subprogram).

15 16.3.2 Local identifiers that are the same as common block names

16 A name that identifies a common block in a scoping unit shall not be used to identify a constant or an
17 intrinsic procedure in that scoping unit. If a local identifier is also the name of a common block, the
18 appearance of that name in any context other than as a common block name in a COMMON or SAVE
19 statement is an appearance of the local identifier.

NOTE 16.5

An intrinsic procedure name may be a common block name in a scoping unit that does not reference
the intrinsic procedure.

20 16.3.3 Function results

21 For each FUNCTION statement or ENTRY statement in a function subprogram, there is a result
22 variable. If there is no RESULT clause, the result variable has the same name as the function being
23 defined; otherwise, the result variable has the name specified in the RESULT clause.

24 16.3.4 Components, type parameters, and bindings

25 A component name has the scope of its derived-type definition. Outside the type definition, it may
26 appear only within a designator of a component of a structure of that type or as a component keyword

1 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,
3 it may appear only as a type parameter keyword in a *derived-type-spec* for the type or as the *type-param-*
4 *name* of a *type-param-inquiry*.

5 The binding name (4.5.5) of a type-bound procedure has the scope of its derived-type definition. Outside
6 of the derived-type definition, it may appear only as the *binding-name* in a procedure reference.

7 A generic binding for which the *generic-spec* is not a *generic-name* has a scope that consists of all scoping
8 units in which an entity of the type is accessible.

9 A component name or binding name may appear only in scoping units in which it is accessible.

10 The accessibility of components and bindings is specified in 4.5.4.7 and 4.5.5.

11 16.3.5 Argument keywords

12 As an argument keyword, a dummy argument name in an internal procedure, module procedure, or
13 an interface body has a scope of the scoping unit of the host of the procedure or interface body. It
14 may appear only in a procedure reference for the procedure of which it is a dummy argument. If the
15 procedure or interface body is accessible in another scoping unit by use association or host association
16 (16.5.1.3, 16.5.1.4), the argument keyword is accessible for procedure references for that procedure in
17 that scoping unit.

18 A dummy argument name in an intrinsic procedure has a scope as an argument keyword of the scoping
19 unit in which the reference to the procedure occurs. As an argument keyword, it may appear only in a
20 procedure reference for the procedure of which it is a dummy argument.

21 16.4 Statement and construct entities

22 A variable that appears as a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array
23 constructor, as a dummy argument in a statement function statement, or as an *index-name* in a FORALL
24 statement is a statement entity. A variable that appears as an *index-name* in a FORALL or DO
25 CONCURRENT construct or as an *associate-name* in a SELECT TYPE or ASSOCIATE construct is
26 a construct entity. A macro local variable is a construct entity. An entity that is declared in a BLOCK
27 construct and is not accessed by use association is a construct entity.

28 If a global or local identifier is the same as that of a construct entity, the identifier is interpreted within
29 the construct as that of the construct entity. Elsewhere in the scoping unit, the identifier is interpreted
30 as the global or local identifier.

31 If a global or local identifier accessible in the scoping unit containing a statement is the same as the
32 name of a statement entity in that statement, the name is interpreted within the scope of the statement
33 entity as that of the statement entity. Elsewhere in the scoping unit, including parts of the statement
34 outside the scope of the statement entity, the name is interpreted as the global or local identifier.

35 If the name of a statement entity is the same as the name of a construct entity and the statement is
36 within the scope of the construct entity, the name is interpreted within the scope of the statement entity
37 as that of the statement entity. Elsewhere in the construct, including parts of the statement outside the
38 scope of the statement entity, the name is interpreted as that of the construct entity.

39 Except for a common block name or a scalar variable name, a global identifier or a local identifier of
40 class (1) (16.3) in the scoping unit that contains a statement shall not be the name of a statement entity
41 of that statement. Within the scope of a statement entity, another statement entity shall not have the

1 same name.

2 The name of a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array constructor
3 has a scope of its *data-implied-do* or *ac-implied-do*. It is a scalar variable that has the type and type
4 parameters that it would have if it were the name of a variable in the scoping unit that includes the
5 DATA statement or array constructor, and this type shall be integer type; it has no other attributes.
6 The appearance of a name as a *data-i-do-variable* of an implied DO in a DATA statement or an *ac-do-*
7 *variable* in an array constructor is not an implicit declaration of a variable whose scope is the scoping
8 unit that contains the statement.

9 The name of a variable that appears as an *index-name* in a FORALL statement or FORALL or DO
10 CONCURRENT construct has a scope of the statement or construct. It is a scalar variable. If *type-spec*
11 appears in *forall-header* it has the specified type and type parameters; otherwise it that has the type and
12 type parameters that it would have if it were the name of a variable in the scoping unit that includes the
13 FORALL, and this type shall be integer type. It has no other attributes. The appearance of a name as
14 an *index-name* in a FORALL statement or FORALL or DO CONCURRENT construct is not an implicit
15 declaration of a variable whose scope is the scoping unit that contains the statement or construct.

16 The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement
17 in which it appears. It is a scalar that has the type and type parameters that it would have if it were the name of a variable
18 in the scoping unit that includes the statement function; it has no other attributes.

19 Except for a common block name or a scalar variable name, a global identifier or a local identifier
20 of class (1) (16.3) in the scoping unit containing a FORALL statement, FORALL construct, or DO
21 CONCURRENT construct shall not be the same as any of its *index-names*. An *index-name* of a
22 contained FORALL statement, FORALL construct, or DO CONCURRENT construct shall not be the
23 same as an *index-name* of any of its containing FORALL or DO CONCURRENT constructs.

24 The associate name of a SELECT TYPE construct has a separate scope for each block of the construct.
25 Within each block, it has the declared type, dynamic type, type parameters, rank, and bounds specified
26 in 8.1.9.2.

27 The associate names of an ASSOCIATE construct have the scope of the block. They have the declared
28 type, dynamic type, type parameters, rank, and bounds specified in 8.1.3.2.

29 The macro local variables of a macro definition have the scope of the macro definition.

30 16.5 Association

31 16.5.1 Name association

32 16.5.1.1 Forms of name association

33 There are five forms of **name association**: argument association, use association, host association,
34 linkage association, and construct association. Argument, use, and host association provide mechanisms
35 by which entities known in one scoping unit may be accessed in another scoping unit.

36 16.5.1.2 Argument association

37 The rules governing argument association are given in Clause 12. As explained in 12.5, execution of a
38 procedure reference establishes an association between an actual argument and its corresponding dummy
39 argument. Argument association may be sequence association (12.5.2.12).

40 The name of the dummy argument may be different from the name, if any, of its associated actual
41 argument. The dummy argument name is the name by which the associated actual argument is known,

1 and by which it may be accessed, in the referenced procedure.

NOTE 16.6

An actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.

2 Upon termination of execution of a procedure reference, all argument associations established by that
3 reference are terminated. A dummy argument of that procedure may be associated with an entirely
4 different actual argument in a subsequent invocation of the procedure.

5 16.5.1.3 Use association

6 **Use association** is the association of names in different scoping units specified by a USE statement. The
7 rules governing use association are given in 11.2.2. They allow for renaming of entities being accessed.
8 Use association allows access in one scoping unit to entities defined in another scoping unit; it remains
9 in effect throughout the execution of the program.

10 16.5.1.4 Host association

11 An internal subprogram, a module subprogram, a submodule subprogram, a module procedure interface
12 body, or a derived-type definition has access to entities from its host via **host association**. An interface
13 body that is not a module procedure interface body has access via host association to the named entities
14 from its host that are made accessible by IMPORT statements in the interface body. The accessed
15 entities are identified by the same identifier and have the same attributes as in the host, except that an
16 accessed entity may have the VOLATILE or ASYNCHRONOUS attribute even if the host entity does
17 not. The accessed entities are named data objects, derived types, abstract interfaces, module procedure
18 interfaces, procedures, generic identifiers, macros, and namelist groups.

19 If an entity that is accessed by use association has the same nongeneric name as a host entity, the host
20 entity is inaccessible by that name. Within the scoping unit, a name that is declared to be an external
21 procedure name by an *external-stmt*, *procedure-declaration-stmt*, or *interface-body*, or that appears as a
22 *module-name* in a *use-stmt* is a global identifier; any entity of the host that has this as its nongeneric
23 name is inaccessible by that name. A name that appears in the scoping unit as

- 24 (1) a *function-name* in a *stmt-function-stmt* or in an *entity-decl* in a *type-declaration-stmt*,
- 25 (2) an *object-name* in an *entity-decl* in a *type-declaration-stmt*, in a *pointer-stmt*, in a *save-stmt*,
26 in an *allocatable-stmt*, or in a *target-stmt*,
- 27 (3) a *type-param-name* in a *derived-type-stmt*,
- 28 (4) a *named-constant* in a *named-constant-def* in a *parameter-stmt*,
- 29 (5) an *array-name* in a *dimension-stmt*,
- 30 (6) a *variable-name* in a *common-block-object* in a *common-stmt*,
- 31 (7) a *proc-pointer-name* in a *common-block-object* in a *common-stmt*,
- 32 (8) the name of a variable that is wholly or partially initialized in a *data-stmt*,
- 33 (9) the name of an object that is wholly or partially equivalenced in an *equivalence-stmt*,
- 34 (10) a *dummy-arg-name* in a *function-stmt*, in a *subroutine-stmt*, in an *entry-stmt*, or in a *stmt-*
35 *function-stmt*,
- 36 (11) a *result-name* in a *function-stmt* or in an *entry-stmt*,
- 37 (12) the name of an entity declared by an interface body,
- 38 (13) an *intrinsic-procedure-name* in an *intrinsic-stmt*,
- 39 (14) a *namelist-group-name* in a *namelist-stmt*,
- 40 (15) the name of a macro in a *define-macro-stmt*,
- 41 (16) a *generic-name* in a *generic-spec* in an *interface-stmt*, or

- 1 (17) the name of a named construct
- 2 is a local identifier in the scoping unit and any entity of the host that has this as its nongeneric name is
- 3 inaccessible by that name by host association. If a scoping unit is the host of a derived-type definition
- 4 or a subprogram that does not define a separate module procedure, the name of the derived type or of
- 5 any procedure defined by the subprogram is a local identifier in the scoping unit; any entity of the host
- 6 that has this as its nongeneric name is inaccessible by that name. Local identifiers of a subprogram are
- 7 not accessible to its host.

NOTE 16.7

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.

- 8 If a host entity is inaccessible only because a local variable with the same name is wholly or partially
- 9 initialized in a `DATA` statement, the local variable shall not be referenced or defined prior to the `DATA`
- 10 statement.
- 11 If a derived-type name of a host is inaccessible, data entities of that type or subobjects of such data
- 12 entities still can be accessible.

NOTE 16.8

An interface body that is not a module procedure interface body accesses by host association only those entities made accessible by `IMPORT` statements.

- 13 If an external or dummy procedure with an implicit interface is accessed via host association, then it
- 14 shall have the `EXTERNAL` attribute in the host scoping unit; if it is invoked as a function in the inner
- 15 scoping unit, its type and type parameters shall be established in the host scoping unit. The type and
- 16 type parameters of a function with the `EXTERNAL` attribute are established in a scoping unit if that
- 17 scoping unit explicitly declares them, invokes the function, accesses the function from a module, or
- 18 accesses the function from its host where its type and type parameters are established.
- 19 If an intrinsic procedure is accessed via host association, then it shall be established to be intrinsic in the
- 20 host scoping unit. An intrinsic procedure is established to be intrinsic in a scoping unit if that scoping
- 21 unit explicitly gives it the `INTRINSIC` attribute, invokes it as an intrinsic procedure, accesses it from a
- 22 module, or accesses it from its host where it is established to be intrinsic.

NOTE 16.9

A host subprogram and an internal subprogram may 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
...

```

NOTE 16.9 (cont.)

CONTAINS

SUBROUTINE INNER_PROC (Q)

USE C ! Not needed

USE B, ONLY: BX ! Entities accessible are BX, IX, and JX
 ! if no other IX or JX
 ! is accessible to INNER_PROC
 ! Q is local to INNER_PROC,
 ! because Q is a dummy argument

USE D, X => DX ! Entities accessible are DX, DY, and DZ
 ! X is local name for DX in INNER_PROC
 ! X and DX denote same entity if no other
 ! entity DX is local to INNER_PROC

USE E, ONLY: EX ! EX is accessible in INNER_PROC, not in program A
 ! EY and EZ are not accessible in INNER_PROC
 ! or in program A

USE G ! FX and GX are accessible in INNER_PROC

...

END SUBROUTINE INNER_PROC

END PROGRAM A

Because program A contains the statement

USE B

all of the entities in module B, except for Q, are accessible in INNER_PROC, even though INNER-
 PROC contains the statement

USE B, ONLY: BX

The USE statement with the ONLY option means that this particular statement brings in only
 the entity named, not that this is the only variable from the module accessible in this scoping unit.

NOTE 16.10

For more examples of host association, see subclause C.12.1.

1 16.5.1.5 Linkage association

2 Linkage association occurs between a module variable that has the BIND attribute and the C variable
 3 with which it interoperates, or between a Fortran common block and the C variable with which it
 4 interoperates (15.4). Such association remains in effect throughout the execution of the program.

5 16.5.1.6 Construct association

6 Execution of a SELECT TYPE statement establishes an association between the selector and the asso-
 7 ciate name of the construct. Execution of an ASSOCIATE statement establishes an association between
 8 each selector and the corresponding associate name of the construct.

- 1 If the selector is allocatable, it shall be allocated; the associate name is associated with the data object
- 2 and does not have the ALLOCATABLE attribute.
- 3 If the selector has the POINTER attribute, it shall be associated; the associate name is associated with
- 4 the target of the pointer and does not have the POINTER attribute.
- 5 If the selector is a variable other than an array section having a vector subscript, the association is
- 6 with the data object specified by the selector; otherwise, the association is with the value of the selector
- 7 expression, which is evaluated prior to execution of the block.
- 8 Each associate name remains associated with the corresponding selector throughout the execution of the
- 9 executed block. Within the block, each selector is known by and may be accessed by the corresponding
- 10 associate name. Upon termination of the construct, the association is terminated.

NOTE 16.11

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*.

11 16.5.2 Pointer association

12 16.5.2.1 General

- 13 Pointer association between a pointer and a target allows the target to be referenced by a reference to the
- 14 pointer. At different times during the execution of a program, a pointer may be undefined, associated
- 15 with different targets, or be disassociated. If a pointer is associated with a target, the definition status
- 16 of the pointer is either defined or undefined, depending on the definition status of the target. If the
- 17 pointer has deferred type parameters or shape, their values are assumed from the target. If the pointer
- 18 is polymorphic, its dynamic type is the dynamic type of the target.

19 16.5.2.2 Pointer association status

- 20 A pointer may have a **pointer association status** of associated, disassociated, or undefined. Its
- 21 association status may change during execution of a program. Unless a pointer is initialized (explicitly
- 22 or by default), it has an initial association status of undefined. A pointer may be initialized to have an
- 23 association status of disassociated or associated.

NOTE 16.12

A pointer from a module program unit may be accessible in a subprogram via use association. Such pointers have a lifetime that is greater than targets that are declared in the subprogram, unless such targets are saved. Therefore, if such a pointer is associated with a local target, there is the possibility that when a procedure defined by the subprogram completes execution, the target will cease to exist, leaving the pointer “dangling”. This part of ISO/IEC 1539 considers such pointers to have an undefined association status. They are neither associated nor disassociated. They shall not be used again in the program until their status has been reestablished. There is no requirement on a processor to be able to detect when a pointer target ceases to exist.

24 16.5.2.2.1 Events that cause pointers to become associated

- 25 A pointer becomes associated when any of the following events occur.

- 26 • The pointer is allocated (6.3.1) as the result of the successful execution of an ALLOCATE statement
- 27 referencing the pointer.

- 1 • The pointer is pointer-assigned to a target (7.2.2) that is associated or is specified with the TAR-
- 2 GET attribute and, if allocatable, is allocated.
- 3 • The pointer is a dummy argument and its corresponding actual argument is not a pointer.
- 4 • The pointer is an ultimate component of an object of a type for which default initialization is
- 5 specified for the component, and the corresponding initializer is an initialization target, and
 - 6 – a procedure is invoked with this object as an actual argument corresponding to a nonpointer
 - 7 nonallocatable dummy argument with INTENT (OUT),
 - 8 – a procedure with this object as an unsaved nonpointer nonallocatable local object that is not
 - 9 accessed by use or host association is invoked, or
 - 10 – this object is allocated.

11 **16.5.2.2.2 Events that cause pointers to become disassociated**

12 A pointer becomes disassociated when

- 13 (1) the pointer is nullified (6.3.2),
- 14 (2) the pointer is deallocated (6.3.3),
- 15 (3) the pointer is pointer-assigned (7.2.2) to a disassociated pointer, or
- 16 (4) the pointer is an ultimate component of an object of a type for which default initialization
- 17 is specified for the component, the corresponding initializer is a reference to the intrinsic
- 18 function NULL, and
 - 19 • a procedure is invoked with this object as an actual argument corresponding to a
 - 20 nonpointer nonallocatable dummy argument with INTENT (OUT),
 - 21 • a procedure with this object as an unsaved nonpointer nonallocatable local object
 - 22 that is not accessed by use or host association is invoked, or
 - 23 • this object is allocated.

24 **16.5.2.2.3 Events that cause the association status of pointers to become undefined**

25 The association status of a pointer becomes undefined when

- 26 (1) the pointer is pointer-assigned to a target that has an undefined association status,
- 27 (2) the target of the pointer is deallocated other than through the pointer,
- 28 (3) the allocation transfer procedure (13.7.124) is executed, the pointer is associated with the
- 29 argument FROM, and an object without the target attribute associated with the argument
- 30 TO,
- 31 (4) execution of a RETURN or END statement causes the pointer's target to become undefined
- 32 (item (3) of 16.6.6),
- 33 (5) termination of a BLOCK construct causes the pointer's target to become undefined (item
- 34 (20) of 16.6.6),
- 35 (6) execution of the host instance of a procedure pointer is completed by execution of a RE-
- 36 TURN or END statement,
- 37 (7) a procedure is terminated by execution of a RETURN or END statement and the pointer
- 38 is declared or accessed in the subprogram that defines the procedure unless the pointer
 - 39 (a) has the SAVE attribute,
 - 40 (b) is in blank common,
 - 41 (c) is in a named common block that is declared in at least one other scoping unit that
 - 42 is in execution,

- 1 (d) is in the scoping unit of a module or submodule,
- 2 (e) is accessed by host association, or
- 3 (f) is the return value of a function declared to have the `POINTER` attribute,
- 4 (8) a `BLOCK` construct is terminated and the pointer is an unsaved local entity that is explicitly
- 5 declared in the `BLOCK`,
- 6 (9) the pointer is an ultimate component of an object, default initialization is not specified
- 7 for the component, and a procedure is invoked with this object as an actual argument
- 8 corresponding to a dummy argument with `INTENT(OUT)`, or
- 9 (10) a procedure is invoked with the pointer as an actual argument corresponding to a pointer
- 10 dummy argument with `INTENT(OUT)`.

11 16.5.2.2.4 Other events that change the association status of pointers

12 When a pointer becomes associated with another pointer by argument association, construct association,
13 or host association, the effects on its association status are specified in 16.5.5.

14 While two pointers are name associated, storage associated, or inheritance associated, if the association
15 status of one pointer changes, the association status of the other changes accordingly.

16 16.5.2.3 Pointer definition status

17 The definition status of a pointer is that of its target. If a pointer is associated with a definable target,
18 the definition status of the pointer may be defined or undefined according to the rules for a variable
19 (16.6).

20 16.5.2.4 Relationship between association status and definition status

21 If the association status of a pointer is disassociated or undefined, the pointer shall not be referenced
22 or deallocated. Whatever its association status, a pointer always may be nullified, allocated, or pointer
23 assigned. A nullified pointer is disassociated. When a pointer is allocated, it becomes associated but
24 undefined. When a pointer is pointer assigned, its association and definition status become those of the
25 specified *data-target* or *proc-target*.

26 16.5.3 Storage association

27 16.5.3.1 General

28 Storage sequences are used to describe relationships that exist among variables, common blocks, and
29 result variables. **Storage association** is the association of two or more data objects that occurs when
30 two or more storage sequences share or are aligned with one or more storage units.

31 16.5.3.2 Storage sequence

32 A **storage sequence** is a sequence of storage units. The **size of a storage sequence** is the number
33 of storage units in the storage sequence. A **storage unit** is a character storage unit, a numeric storage
34 unit, a file storage unit(9.2.4), or an unspecified storage unit. The sizes of the numeric storage unit, the
35 character storage unit and the file storage unit are the values of constants in the `ISO_FORTRAN_ENV`
36 intrinsic module (13.8.2).

37 In a storage association context

- 38 (1) a nonpointer scalar object of type default integer, default real, default logical, or default
39 bits occupies a single **numeric storage unit**,

- (2) a nonpointer scalar object of type double precision real or default complex occupies two contiguous numeric storage units,
- (3) A nonpointer scalar object of type bits with a kind type parameter that is an integer multiple, N , of the size of a numeric storage unit occupies N contiguous numeric storage units. A nonpointer scalar object of type bits with a kind type parameter that is not an integer multiple of the size of a numeric storage unit occupies an unspecified storage unit that is different for each such kind value.
- (4) a nonpointer scalar object of type default character and character length len occupies len contiguous **character storage units**,
- (5) a nonpointer scalar object of type character with the C character kind (15.2.2) and character length len occupies len contiguous **unspecified storage units**,
- (6) a nonpointer scalar object of sequence type with no type parameters occupies a sequence of storage sequences corresponding to the sequence of its ultimate components,
- (7) a nonpointer scalar object of any type not specified in items (1)-(6) 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 (5),
- (8) a nonpointer array occupies a sequence of contiguous storage sequences, one for each array element, in array element order (6.2.2.1), and
- (9) a 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 pointer that has the CONTIGUOUS attribute occupies a storage unit that is different from that of a pointer that does not have the CONTIGUOUS attribute.

A sequence of storage sequences forms a storage sequence. The order of the storage units in such a composite storage sequence is that of the individual storage units in each of the constituent storage sequences taken in succession, ignoring any zero-sized constituent sequences.

Each common block has a storage sequence (5.7.2.2).

NOTE 16.13

For a BITS value, the order of its storage units is processor dependent. For example,

```

BITS X(2)
BITS(KIND(X)*2) Y, ZLE, ZBE
...
X = TRANSFER (Y, X)
ZBE = X(1)//X(2)
ZLE = X(2)//X(1)
!
! On some processors Y==ZLE is true and on other processors Y==ZBE is true.
```

NOTE 16.14

A nonpointer nonallocatable scalar BITS object with a KIND value that is not an integer multiple of the size of a numeric storage unit in bits might be stored in a memory region larger than the minimum required to represent the value. For example, if BITS_KIND(x) has the value 13, the storage size for x might be 16 bits. Each element of a BITS array occupies the same size memory region as a scalar BITS object of the same kind.

16.5.3.3 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
 2 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
 3 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 $1 \leq j + k \leq$
 4 $\text{size of } s_2$.

5 Storage association also is defined between two zero-sized storage sequences, and between a zero-sized
 6 storage sequence and a storage unit. A zero-sized storage sequence in a sequence of storage sequences is
 7 storage associated with its successor, if any. If the successor is another zero-sized storage sequence, the
 8 two sequences are storage associated. If the successor is a nonzero-sized storage sequence, the zero-sized
 9 sequence is storage associated with the first storage unit of the successor. Two storage units that are
 10 each storage associated with the same zero-sized storage sequence are the same storage unit.

NOTE 16.15

Zero-sized objects may occur in a storage association context as the result of changing a parameter.
 For example, a program might contain the following declarations:

```
INTEGER, PARAMETER :: PROBSIZE = 10
INTEGER, PARAMETER :: ARRAYSIZE = PROBSIZE * 100
REAL, DIMENSION (ARRAYSIZE) :: X
INTEGER, DIMENSION (ARRAYSIZE) :: IX
...
COMMON / EXAMPLE / A, B, C, X, Y, Z
EQUIVALENCE (X, IX)
...
```

If the first statement is subsequently changed to assign zero to PROBSIZE, the program still will conform to the standard.

11 16.5.3.4 Association of scalar data objects

12 Two scalar data objects are storage associated if their storage sequences are storage associated. Two
 13 scalar entities are **totally associated** if they have the same storage sequence. Two scalar entities are
 14 **partially associated** if they are associated without being totally associated.

15 The definition status and value of a data object affects the definition status and value of any storage
 16 associated entity. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement
 17 can cause storage association of storage sequences.

18 An EQUIVALENCE statement causes storage association of data objects only within one scoping unit,
 19 unless one of the equivalenced entities is also in a common block (5.7.1.2 and 5.7.2.2).

20 COMMON statements cause data objects in one scoping unit to become storage associated with data
 21 objects in another scoping unit.

22 A common block is permitted to contain a sequence of differing storage units. All scoping units that
 23 access named common blocks with the same name shall specify an identical sequence of storage units.
 24 Blank common blocks may be declared with differing sizes in different scoping units. For any two blank
 25 common blocks, the initial sequence of storage units of the longer blank common block shall be identical
 26 to the sequence of storage units of the shorter common block. If two blank common blocks are the same
 27 length, they shall have the same sequence of storage units.

28 An ENTRY statement in a function subprogram causes storage association of the result variables.

Partial association shall exist only between

- 1 (1) an object of default character or character sequence type and an object of default character
 2 or character sequence type, or
 3 (2) an object of default complex, double precision real, or numeric sequence type and an object
 4 of default integer, default real, default logical, double precision real, default complex, or
 5 numeric sequence type.
- 6 For noncharacter entities, partial association may occur only through the use of COMMON, EQUIV-
 7 ALENCE, or ENTRY statements. For character entities, partial association may occur only through
 8 argument association or the use of COMMON or EQUIVALENCE statements.

NOTE 16.16

In the example:

```
REAL A (4), B
COMPLEX C (2)
DOUBLE PRECISION D
EQUIVALENCE (C (2), A (2), B), (A, D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:

Storage unit	1	2	3	4	5
	----C(1)----		---C(2)----		
		A(1)	A(2)	A(3)	A(4)
			--B--		
			-----D-----		

A (2) and B are totally associated. The following are partially associated: A (1) and C (1), A (2) and C (2), A (3) and C (2), B and C (2), A (1) and D, A (2) and D, B and D, C (1) and D, and C (2) and D. Although C (1) and C (2) are each storage associated with D, C (1) and C (2) are not storage associated with each other.

- 9 Partial association of character entities occurs when some, but not all, of the storage units of the entities
 10 are the same.

NOTE 16.17

In the example:

```
CHARACTER A*4, B*4, C*3
EQUIVALENCE (A (2:3), B, C)
```

A, B, and C are partially associated.

- 11 A storage unit shall not be explicitly initialized more than once in a program. Explicit initialization
 12 overrides default initialization, and default initialization for an object of derived type overrides default
 13 initialization for a component of the object (4.5.2). Default initialization may be specified for a storage
 14 unit that is storage associated provided the objects supplying the default initialization are of the same
 15 type and type parameters, and supply the same value for the storage unit.

16 16.5.4 Inheritance association

1 Inheritance association occurs between components of the parent component and components inherited
2 by type extension into an extended type (4.5.7.2). This association is persistent; it is not affected by the
3 accessibility of the inherited components.

4 **16.5.5 Establishing associations**

5 When an association is established between two entities by argument association, host association,
6 or construct association, certain characteristics of the **associating entity** become those of the **pre-**
7 **existing** entity.

8 For argument association, if the dummy argument is not a pointer and the corresponding actual argument
9 is a pointer, the pre-existing entity is the target of that pointer. Otherwise the pre-existing entity is the
10 corresponding actual argument. In either case, the associating entity is the dummy argument.

11 For host association, the associating entity is the entity in the host scoping unit and the pre-existing
12 entity is the entity in the contained scoping unit. If an internal procedure is invoked via a dummy
13 procedure or procedure pointer, the pre-existing entity that participates in the association is the one
14 from the host instance. Otherwise, if the host scoping unit is a recursive procedure, the pre-existing entity
15 that participates in the association is the one from the innermost subprogram instance that invoked,
16 directly or indirectly, the contained procedure.

17 For construct association, the associating entity is identified by the associate name and the pre-existing
18 entity is the selector.

19 When an association is established by argument association, host association, or construct association,
20 the following applies.

- 21 • If the associating entity has the POINTER attribute, its pointer association status becomes the
22 same as that of the pre-existing entity. If the pre-existing entity has a pointer association status of
23 associated, the associating entity becomes pointer associated with the same target and, if they are
24 arrays, the bounds of the associating entity become the same as those of the pre-existing entity.
- 25 • If the associating entity has the ALLOCATABLE attribute, its allocation status becomes the same
26 as that of the pre-existing entity. If the pre-existing entity is allocated, the bounds (if it is an array),
27 values of deferred type parameters, definition status, and value (if it is defined) become the same
28 as those of the pre-existing entity. If the associating entity is polymorphic and the pre-existing
29 entity is allocated, the dynamic type of the associating entity becomes the same as that of the
30 pre-existing entity.
- 31 • If the associating entity is neither a pointer nor allocatable, its definition status, value (if it is
32 defined), and dynamic type (if it is polymorphic) become the same as those of the pre-existing
33 entity. If the entities are arrays and the association is not argument association, the bounds of the
34 associating entity become the same as those of the pre-existing entity.

35 **16.6 Definition and undefinition of variables**

36 **16.6.1 Definition of objects and subobjects**

37 A variable may be defined or may be undefined and its definition status may change during execution of
38 a program. An action that causes a variable to become undefined does not imply that the variable was
39 previously defined. An action that causes a variable to become defined does not imply that the variable
40 was previously undefined.

1 Arrays, including sections, and variables of derived, character, or complex type are objects that consist of
2 zero or more subobjects. Associations may be established between variables and subobjects and between
3 subobjects of different variables. These subobjects may become defined or undefined.

4 An array is defined if and only if all of its elements are defined.

5 A derived-type scalar object is defined if and only if all of its nonpointer components are defined.

6 A complex or character scalar object is defined if and only if all of its subobjects are defined.

7 If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

8 **16.6.2 Variables that are always defined**

9 Zero-sized arrays and zero-length strings are always defined.

10 **16.6.3 Variables that are initially defined**

11 The following variables are initially defined:

- 12 (1) variables specified to have initial values by DATA statements;
- 13 (2) variables specified to have initial values by type declaration statements;
- 14 (3) nonpointer default-initialized subcomponents of variables that do not have the ALLOCAT-
15 ABLE or POINTER attribute, and are saved, local variables of a main program, or declared
16 in a MODULE or BLOCK DATA scoping unit;
- 17 (4) pointers specified to be initially associated with a variable that is initially defined;
- 18 (5) variables that are always defined;
- 19 (6) variables with the BIND attribute that are initialized by means other than Fortran.

NOTE 16.18

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.

20 **16.6.4 Variables that are initially undefined**

21 All other variables are initially undefined.

22 **16.6.5 Events that cause variables to become defined**

23 Variables become defined by the following events.

- 24 (1) Execution of an intrinsic assignment statement other than a masked array assignment or
25 FORALL assignment statement causes the variable that precedes the equals to become
26 defined.

- (2) Execution of a masked array assignment or FORALL assignment statement might cause some or all of the array elements in the assignment statement to become defined (7.2.3).
- (3) As execution of an input statement proceeds, each variable that is assigned a value from the input file becomes defined at the time that data is transferred to it. (See (4) in 16.6.6.) Execution of a WRITE statement whose unit specifier identifies an internal file causes each record that is written to become defined.
- (4) Execution of a DO statement causes the DO variable, if any, to become defined.
- (5) Beginning of execution of the action specified by an *io-implied-do* in a synchronous input/output statement causes the *do-variable* to become defined.
- (6) A reference to a procedure causes the entire dummy argument data object to become defined if the dummy argument does not have INTENT(OUT) and the entire corresponding actual 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 corresponding actual argument is defined.
- (7) Execution of an input/output statement containing an IOSTAT= specifier causes the specified integer variable to become defined.
- (8) Execution of a synchronous READ statement containing a SIZE= specifier causes the specified integer variable to become defined.
- (9) Execution of a wait operation 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) When a dummy argument of type bits that is associated with an effective argument of a different type becomes defined, the effective argument becomes defined.

J3 internal note

Unresolved Technical Issue 076

We still have a “bits define too much” situation.

I will draw people’s attention to one of the *primary* goals of the Fortran standard, which is to promote portability. This runs directly counter to that goal.

Since bits used in this manner are inherently nonportable, we need to limit them, not throw out the portability baby.

Furthermore, “invalid values”, i.e. values that don’t act consistently, or indeed crash the program, are not limited to LOGICALs.

We really *ought not* to go further than C here. C explicitly acknowledges that there may be “trap representations”; trap in this context means *don’t fall into it!* We ought not to attempt to require implementations not to have trap representations!!!! Many processors in the past have had trap representations for REALs; even in the recent past it has been common, e.g. on processors without full IEEE, representations of unsupported IEEE values would just trap (that’s why we put IEEE_OTHER_VALUE into IEEE_CLASS).

There have also been processors (in the not-too-distant past) which have had trap representations for INTEGERS. And even now, many processors have LOGICAL representations that don’t act like values.

Bit-twiddling the guts of a Fortran data type is *inherently* processor-dependent. We should have the guts to say that.

Furthermore, it provides maximum traction to the feature, with the smallest effort in standards-writing.

I.e. replace the above with

(16)When a dummy argument of type bits that is associated with an effective argument of a different type becomes defined, whether the effective argument becomes defined is processor dependent.

- 1 (17) Execution of a statement with a STAT= specifier causes the variable specified by the STAT=
- 2 specifier to become defined.
- 3 (18) If an error condition occurs during execution of a statement that has an ERRMSG= specifier,
- 4 the variable specified by the ERRMSG= specifier becomes defined.
- 5 (19) Allocation of a zero-sized array causes the array to become defined.
- 6 (20) Allocation of an object that has a nonpointer default-initialized subcomponent causes that
- 7 subcomponent to become defined.
- 8 (21) Invocation of a procedure causes any automatic object of zero size in that procedure to
- 9 become defined.
- 10 (22) Execution of a pointer assignment statement that associates a pointer with a target that is
- 11 defined causes the pointer to become defined.
- 12 (23) Invocation of a procedure that contains an unsaved nonpointer nonallocatable local variable
- 13 causes all nonpointer default-initialized subcomponents of the object to become defined.
- 14 (24) Invocation of a procedure that has a nonpointer nonallocatable INTENT (OUT) dummy
- 15 argument causes all nonpointer default-initialized subcomponents of the dummy argument
- 16 to become defined.
- 17 (25) Invocation of a nonpointer function of a derived type causes all nonpointer default-initialized
- 18 subcomponents of the function result to become defined.
- 19 (26) In a FORALL or DO CONCURRENT construct, the *index-name* becomes defined when
- 20 the *index-name* value set is evaluated.
- 21 (27) An object with the VOLATILE attribute that is changed by a means not specified by the
- 22 program becomes defined (see 5.3.18).
- 23 (28) Execution of the BLOCK statement of a BLOCK construct that has an unsaved nonpointer
- 24 nonallocatable local variable causes all nonpointer default-initialized subcomponents of the

1 variable to become defined.

2 (29) Execution of an OPEN statement containing a NEWUNIT= specifier causes the specified
3 integer variable to become defined.

4 **16.6.6 Events that cause variables to become undefined**

5 Variables become undefined by the following events.

6 (1) When a variable of a given type becomes defined, all associated variables of different type
7 become undefined. However, when a variable of type default real is partially associated with
8 a variable of type default complex, the complex variable does not become undefined when
9 the real variable becomes defined and the real variable does not become undefined when
10 the complex variable becomes defined. When a variable of type default complex is partially
11 associated with another variable of type default complex, definition of one does not cause
12 the other to become undefined. When a dummy argument of type bits is associated with
13 an actual argument of a different type, definition of the dummy argument does not cause
14 the actual argument to become undefined.

15 (2) If the evaluation of a function would cause a variable to become defined and if a reference
16 to the function appears in an expression in which the value of the function is not needed to
17 determine the value of the expression, the variable becomes undefined when the expression
18 is evaluated.

19 (3) When execution of an instance of a subprogram completes,

20 (a) its unsaved local variables become undefined, and

21 (b) unsaved variables in a named common block that appears in the subprogram become
22 undefined if they have been defined or redefined, unless another active scoping unit
23 is referencing the common block or the common block is declared in a module or
24 submodule.

25 (4) When an error condition or end-of-file condition occurs during execution of an input state-
26 ment, all of the variables specified by the input list or namelist group of the statement
27 become undefined.

28 (5) When an error condition, end-of-file condition, or end-of-record condition occurs during
29 execution of an input/output statement and the statement contains any *io-implied-dos*, all
30 of the *do-variables* in the statement become undefined (9.10).

31 (6) Execution of a direct access input statement that specifies a record that has not been written
32 previously causes all of the variables specified by the input list of the statement to become
33 undefined.

34 (7) Execution of an INQUIRE statement might cause the NAME=, RECL=, and NEXTREC=
35 variables to become undefined (9.9).

36 (8) When a character storage unit becomes undefined, all associated character storage units
37 become undefined.

38 When a numeric storage unit becomes undefined, all associated numeric storage units be-
39 come undefined unless the undefinition is a result of defining an associated numeric storage
40 unit of different type (see (1) above).

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

43 When an unspecified storage unit becomes undefined, all associated unspecified storage units
44 become undefined.

45 (9) When an allocatable entity is deallocated, it becomes undefined.

46 (10) When the allocation transfer procedure (13.5.17) causes the allocation status of an allocat-
47 able entity to become unallocated, the entity becomes undefined.

- (11) Successful execution of an ALLOCATE statement for a nonzero-sized object that has a subcomponent for which default initialization has not been specified causes the subcomponent to become undefined.
- (12) 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.
- (13) When a procedure is invoked
 - (a) an optional dummy argument that is not associated with an actual argument becomes undefined,
 - (b) a dummy argument with INTENT (OUT) becomes undefined except for any non-pointer default-initialized subcomponents of the argument,
 - (c) an actual argument associated with 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 actual argument is undefined, and
 - (e) the result variable of a function becomes undefined except for any of its nonpointer default-initialized subcomponents.
- (14) When the association status of a pointer becomes undefined or disassociated (16.5.2.2.2-16.5.2.2.3), the pointer becomes undefined.
- (15) When the execution of a FORALL or DO CONCURRENT construct completes, the *index-names* become undefined.
- (16) When a DO CONCURRENT construct terminates, a variable that is defined or becomes undefined during more than one iteration of the construct becomes undefined.
- (17) Execution of an asynchronous READ statement causes all of the variables specified by the input list or SIZE= specifier to become undefined. Execution of an asynchronous namelist READ statement causes any variable in the namelist group to become undefined if that variable will subsequently be defined during the execution of the READ statement or the corresponding WAIT operation.
- (18) When execution of a RETURN or END statement causes a variable to become undefined, any variable of type C_PTR becomes undefined if its value is the C address of any part of the variable that becomes undefined.
- (19) When a variable with the TARGET attribute is deallocated, any variable of type C_PTR becomes undefined if its value is the C address of any part of the variable that is deallocated.

J3 internal note

Unresolved Technical Issue 192

C_PTR and C_FUNPTR undefined events defective.

The above events are defective, in that they omit events which cause pointers to go undefined (16.5.2.2.3).

Some of these events are new to F2008 (hello BLOCK), at least one existed in F2003 (hello MOVE_ALLOC).

BTW, the wording above ("any part of") is unnecessary and ought to be deleted.

It would be nice to reword this in a future-proof way, e.g. replace both the existing items (and the new item about C_FUNPTR) with

(18) When an event occurs that would cause the association status of a pointer associated with the target of a C_PTR or C_FUNPTR variable to become undefined (16.5.2.2.3), the variable of type C_PTR or C_FUNPTR becomes undefined.

The above suggestion has some potential ambiguities so might need a bit more care in wording. If that's too hard, just re-list all the relevant events from 16.5.2.2.3 here.

- 1 (20) When a BLOCK construct terminates, its unsaved local variables become undefined.
 2 (21) When execution of the host instance of the target of a variable of type C_FUNPTR is
 3 completed by execution of a RETURN or END statement, the variable becomes undefined.

NOTE 16.19

Execution of a defined assignment statement may leave all or part of the variable undefined.

4 **16.6.7 Variable definition context**

5 Some variables are prohibited from appearing in a syntactic context that would imply definition or
 6 undefined of the variable (5.3.9, 5.3.14, 12.7). The following are the contexts in which the appearance
 7 of a variable implies such definition or undefined of the variable:

- 8 • the *variable* of an *assignment-stmt*;
- 9 • a *pointer-object* in a *nullify-stmt*;
- 10 • a *data-pointer-object* or *proc-pointer-object* in a *pointer-assignment-stmt*;
- 11 • a *do-variable* in a *do-stmt* or *io-implied-do*;
- 12 • an *input-item* in a *read-stmt*;
- 13 • a *variable-name* in a *namelist-stmt* if the *namelist-group-name* appears in a NML= specifier in a
 14 *read-stmt*;
- 15 • an *internal-file-variable* in a *write-stmt*;
- 16 • an IOSTAT=, SIZE=, or IOMSG= specifier in an input/output statement;
- 17 • a specifier in an INQUIRE statement other than FILE=, ID=, and UNIT=;
- 18 • a NEWUNIT= specifier in an OPEN statement;
- 19 • a READY= specifier in a QUERY statement;
- 20 • a *stat-variable*, *allocate-object*, or *errmsg-variable*;
- 21 • an actual argument in a reference to a procedure with an explicit interface if the associated dummy
 22 argument has the INTENT(OUT) or INTENT(INOUT) attribute;
- 23 • a *variable* that is the *selector* in a SELECT TYPE or ASSOCIATE construct if the associate name
 24 of that construct appears in a variable definition context.

25 **16.6.8 Pointer association context**

26 Some pointers are prohibited from appearing in a syntactic context that would imply alteration of the
 27 pointer association status (16.5.2.2, 5.3.9, 5.3.14). The following are the contexts in which the appearance
 28 of a pointer implies such alteration of its pointer association status:

- 29 • a *pointer-object* in a *nullify-stmt*;
- 30 • a *data-pointer-object* or *proc-pointer-object* in a *pointer-assignment-stmt*;
- 31 • an *allocate-object* in an *allocate-stmt* or *deallocate-stmt*;

- 1 • an actual argument in a reference to a procedure if the associated dummy argument is a pointer
- 2 with the INTENT (OUT) or INTENT (INOUT) attribute.

- 3 If a reference to a function appears in a variable definition context the result of the function reference
- 4 shall be a pointer that is associated with a definable target. That target is the variable that becomes
- 5 defined or undefined.

Annex A

(Informative)

Glossary of technical terms

The following is a list of the principal technical terms used in this part of ISO/IEC 1539 and their definitions. A reference in parentheses immediately after a term is to the clause or subclause where the term is defined or explained. The wording of a definition here is not necessarily the same as its normative definition.

abstract type (4.5.7) : A type that has the ABSTRACT attribute. A nonpolymorphic object shall not be declared to be of abstract type. A polymorphic object shall not be constructed or allocated to have a dynamic abstract type.

actual argument (12, 12.5.2) : An expression, a variable, a procedure, or an alternate return specifier that is specified in a procedure reference.

allocatable variable (5.3.3) : A variable having the ALLOCATABLE attribute. It may be referenced or defined only when it is allocated. If it is an array, it has a shape only when it is allocated. It may be a named variable or a structure component.

argument (12) : An actual argument or a dummy argument.

argument association (16.5.1.2) : The relationship between an actual argument and a dummy argument during the execution of a procedure reference.

array (2.4.5) : A set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. It may be a named array, an array section, a structure component, a function value, or an expression. Its rank is at least one. Note that in FORTRAN 77, arrays were always named and never constants.

array element (2.4.5, 6.2.2) : One of the scalar data that make up an array that is either named or is a structure component.

array pointer (5.3.7.4) : A pointer to an array.

array section (2.4.5, 6.2.2.2) : A subobject that is an array and is not a structure component.

assignment statement (7.2.1.1) : A statement that evaluates an expression and assigns its value to a variable.

associate name (8.1.3.1) : The name of the construct entity with which a selector of a SELECT TYPE or ASSOCIATE construct is associated within the construct.

association (16.5) : Name association, pointer association, storage association, or inheritance association.

assumed-shape array (5.3.7.3) : A nonpointer dummy array that takes its shape from the associated actual argument.

assumed-size array (5.3.7.5) : A dummy array whose size is assumed from the associated actual argument. Its last upper bound is specified by an asterisk.

attribute (5) : A property of an entity that determines its uses.

- 1 **automatic data object** (5.2) : A data object that is a local entity of a subprogram, that is not a dummy
 2 argument, and that has a length type parameter or array bound that is specified by an expression that
 3 is not an initialization expression.
- 4 **base type** (4.5.7) : An extensible type that is not an extension of another type.
- 5 **belong** (8.1.7.5.3, 8.1.7.5.4) : If an EXIT or a CYCLE statement contains a construct name, the
 6 statement **belongs** to the construct using that name. Otherwise, it **belongs** to the innermost DO
 7 construct in which it appears.
- 8 **binding label** (15.5.2, 15.4.2) : A value of type default character that uniquely identifies how a variable,
 9 common block, subroutine, or function is known to a companion processor.
- 10 **bits compatible** (12.5.2.4) : An entity is bits compatible with another entity if and only if one is of
 11 type bits, the other is of type bits, integer, real, complex, or logical, and scalar entities of these types
 12 have the same size expressed in bits.
- 13 **block** (8.1) : A sequence of executable constructs embedded in another executable construct, bounded
 14 by statements that are particular to the construct, and treated as an integral unit.
- 15 **block data program unit** (11.3) : A program unit that provides initial values for data objects in
 16 named common blocks.
- 17 **bounds** (5.3.7.2) : For a named array, the limits within which the values of the subscripts of its array
 18 elements shall lie.
- 19 **character length parameter** (2.4.1.1) : The type parameter that specifies the number of characters
 20 for an entity of type character.
- 21 **character storage unit** (16.5.3.2) : The unit of storage for holding a scalar that is not a pointer and
 22 is of type default character and character length one.
- 23 **character string** (4.4.5) : A sequence of characters numbered from left to right 1, 2, 3, ...
- 24 **characteristics** (12.3) :
- 25 Of a procedure, its classification as a function or subroutine, whether it is pure, whether it is elemental,
 26 whether it has the BIND attribute, the value of its binding label, the characteristics of its dummy
 27 arguments, and the characteristics of its function result if it is a function.
- 28 Of a dummy argument, whether it is a data object, is a procedure, is a procedure pointer, is an asterisk
 29 (alternate return indicator), or has the OPTIONAL attribute.
- 30 Of a dummy data object, its type, type parameters, shape, the exact dependence of an array bound
 31 or type parameter on other entities, intent, whether it is optional, whether it is a pointer or a target,
 32 whether it is allocatable, whether it has the VALUE, ASYNCHRONOUS, or VOLATILE attributes,
 33 whether it is polymorphic, and whether the shape, size, or a type parameter is assumed.
- 34 Of a dummy procedure or procedure pointer, whether the interface is explicit, the characteristics of the
 35 procedure if the interface is explicit, and whether it is optional.
- 36 Of a function result, its type, type parameters, which type parameters are deferred, whether it is poly-
 37 morphic, whether it is a pointer or allocatable, whether it is a procedure pointer, rank if it is a pointer
 38 or allocatable, shape if it is not a pointer or allocatable, the exact dependence of an array bound or type
 39 parameter on other entities, and whether the character length is assumed.
- 40 **class** (4.3.1.3) : A class named N is the set of types extended from the type named N.

- 1 **co-array** (2.4.6) A data entity that has nonzero co-rank. A non-dummy co-array has the same shape
2 on every image and its values may be referenced or defined by any image.
- 3 **co-bounds** (5.3.7) : For a co-array, the limits within which the values of the co-subscripts of its co-
4 indexed objects shall lie.
- 5 **co-indexed object** (2.4.6) : An object whose designator includes an *image-selector*.
- 6 **co-rank** (2.4.6) : The number of co-dimensions of a co-array.
- 7 **co-subscript** (6.2.3) : One of the list of scalar integer expressions in an *image-selector*.
- 8 **collating sequence** (4.4.5.4) : An ordering of all the different characters of a particular kind type
9 parameter.
- 10 **collective subroutine** (13.1) : An intrinsic subroutine that has an argument of type IMAGE_TEAM.
- 11 **common block** (5.7.2) : A block of physical storage that may be accessed by any of the scoping units
12 in a program.
- 13 **companion processor** (2.5.10): A mechanism by which global data and procedures may be referenced
14 or defined. It may be a mechanism that references and defines such entities by means other than Fortran.
15 The procedures can be described by a C function prototype.
- 16 **component** (4.5) : A constituent of a derived type.
- 17 **component order** (4.5.4.6) : The ordering of the components of a derived type that is used for intrinsic
18 formatted input/output and for structure constructors.
- 19 **conformable** (2.4.5) : Two arrays are **conformable** if they have the same shape. A scalar is con-
20 formable with any array.
- 21 **conformance** (1.5) : A program conforms to this part of ISO/IEC 1539 if it uses only those forms
22 and relationships described therein and if the program has an interpretation according to this part of
23 ISO/IEC 1539. A program unit conforms to this part of ISO/IEC 1539 if it can be included in a program
24 in a manner that allows the program to be standard-conforming. A processor conforms to this part of
25 ISO/IEC 1539 if it executes standard-conforming programs in a manner that fulfills the interpretations
26 prescribed in this part of ISO/IEC 1539 and contains the capability of detection and reporting as listed
27 in 1.5.
- 28 **connect team** (9.4.5.17) : The set of images that are permitted to reference a particular external
29 input/output unit.
- 30 **connected** (9.4.3) :
- 31 For an external unit, the property of referring to an external file.
- 32 For an external file, the property of being referred to by an external unit.
- 33 **constant** (2.4.3.1.2) : A data object whose value shall not change during execution of a program. It
34 may be a named constant or a literal constant.
- 35 **construct** (7.2.3, 7.2.4, 8.1) : A sequence of statements starting with an ASSOCIATE, BLOCK, DO,
36 FORALL, IF, SELECT CASE, SELECT TYPE, or WHERE statement and ending with the correspond-
37 ing terminal statement.
- 38 **construct association** (16.5.1.6) : The association between the selector of an ASSOCIATE or SELECT
39 TYPE construct and the associate name.

- 1 **construct entity** (16) : An entity defined by a lexical token whose scope is a construct.
- 2 **control mask** (7.2.3) : In a WHERE statement or construct, an array of type logical whose value
3 determines which elements of an array, in a *where-assignment-stmt*, will be defined.
- 4 **data** : Plural of datum.
- 5 **data entity** (2.4.3) : A data object, the result of the evaluation of an expression, or the result of the
6 execution of a function reference (called the function result). A data entity has a type (either intrinsic
7 or derived) and has, or may have, a data value (the exception is an undefined variable). Every data
8 entity has a rank and is thus either a scalar or an array.
- 9 **data object** (2.4.3.1) : A data entity that is a constant, a variable, or a subobject of a constant.
- 10 **data type** (4) : See type.
- 11 **datum** : A single quantity that may have any of the set of values specified for its type.
- 12 **decimal symbol** (9.9.2.6, 10.6, 10.8.8) : The character that separates the whole and fractional parts in
13 the decimal representation of a real number in a file. By default the decimal symbol is a decimal point
14 (also known as a period). The current decimal symbol is determined by the current decimal edit mode.
- 15 **declared type** (4.3.1.3, 7.1.9) : The type that a data entity is declared to have. May differ from the
16 type during execution (the dynamic type) for polymorphic data entities.
- 17 **default initialization** (4.5) : If initialization is specified in a type definition, an object of the type will
18 be automatically initialized. Nonpointer components may be initialized with values by default; pointer
19 components may be initially disassociated by default. Default initialization is not provided for objects
20 of intrinsic type.
- 21 **default-initialized** (4.5.4.5) : A subcomponent is said to be default-initialized if it will be initialized
22 by default initialization.
- 23 **deferred binding** (4.5.5) : A binding with the DEFERRED attribute. A deferred binding shall appear
24 only in an abstract type definition (4.5.7).
- 25 **deferred type parameter** (4.3) : A length type parameter whose value is not specified in the decla-
26 ration of an object, but instead is specified when the object is allocated or pointer-assigned.
- 27 **definable** (2.5.5) : A variable is **definable** if its value may be changed by the appearance of its
28 designator on the left of an assignment statement. An allocatable variable that has not been allocated
29 is an example of a data object that is not definable. An example of a subobject that is not definable is
30 C (I) when C is an array that is a constant and I is an integer variable.
- 31 **defined** (2.5.5) : For a data object, the property of having or being given a valid value.
- 32 **defined assignment statement** (7.2.1.4, 12.4.3.4.3) : An assignment statement that is not an intrinsic
33 assignment statement; it is defined by a subroutine and a generic interface that specifies ASSIGNMENT
34 (=).
- 35 **defined operation** (7.1.6, 12.4.3.4.2) : An operation that is not an intrinsic operation and is defined
36 by a function that is associated with a generic identifier.
- 37 **deleted feature** (1.8) : A feature in a previous Fortran standard that is considered to have been
38 redundant and largely unused. See B.1 for a list of features that are in a previous Fortran standard, but
39 are not in this part of ISO/IEC 1539. A feature designated as an obsolescent feature in this edition of
40 this part of ISO/IEC 1539 may become a deleted feature in the next revision.

- 1 **derived type** (2.4.1.2, 4.5) : A type whose data have components, each of which is either of intrinsic
2 type or of another derived type.
- 3 **designator** (2.5.1) : A name, followed by zero or more subobject selectors.
- 4 **disassociated** (2.4.7) : A disassociated pointer is not associated with a target. A pointer is disassociated
5 following execution of a NULLIFY statement, following pointer assignment with a disassociated pointer,
6 by default initialization, or by explicit initialization. A data pointer may also be disassociated by
7 execution of a DEALLOCATE statement.
- 8 **dummy argument** (12, 12.6.2.2, 12.6.2.3, 12.6.2.6, 12.6.4) : An entity by which an associated actual
9 argument is accessed during execution of a procedure.
- 10 **dummy array** : A dummy argument that is an array.
- 11 **dummy data object** (12.3.2.2, 12.5.2.5-12.5.2.8) : A dummy argument that is a data object.
- 12 **dummy procedure** (12.2.2.3) : A dummy argument that is specified or referenced as a procedure.
- 13 **dynamic type** (4.3.1.3, 7.1.9) : The type of a data entity during execution of a program. The dynamic
14 type of a data entity that is not polymorphic is the same as its declared type.
- 15 **effective item** (9.5.3) : A scalar object resulting from expanding an input/output list according to the
16 rules in 9.5.3.
- 17 **elemental** (2.4.5, 7.2.1.4, 12.8) : An adjective applied to an operation, procedure, or assignment state-
18 ment that is applied independently to elements of an array or corresponding elements of a set of con-
19 formable arrays and scalars.
- 20 **entity** : The term used for any of the following: an abstract interface, common block, construct, data
21 entity, external unit, generic interface, image, macro, namelist group, operator, procedure, program unit,
22 statement function, statement label, or type.
- 23 **executable construct** (2.1) : An action statement (R214) or an ASSOCIATE, CASE, DO, FORALL,
24 IF, SELECT TYPE, or WHERE construct.
- 25 **executable statement** (2.3.3) : An instruction to perform or control one or more computational
26 actions.
- 27 **explicit initialization** (5.2) : Explicit initialization may be specified for objects of intrinsic or derived
28 type in type declaration statements or DATA statements. An object of a derived type that specifies
29 default initialization shall not appear in a DATA statement.
- 30 **explicit interface** (12.4.2) : If a procedure has an explicit interface at the point of a reference to it, the
31 processor is able to verify that the characteristics of the reference and declaration are related as required
32 by this part of ISO/IEC 1539. A procedure has an explicit interface if it is an internal procedure, a
33 module procedure, an intrinsic procedure, an external procedure that has an interface body, a procedure
34 reference in its own scoping unit, or a dummy procedure that has an interface body.
- 35 **explicit-shape array** (5.3.7.2) : A named array that is declared with explicit bounds.
- 36 **expression** (2.4.3.2, 7.1) : A sequence of operands, operators, and parentheses (R722). It may be a
37 variable, a constant, a function reference, or may represent a computation.
- 38 **extended type** (4.5.7) : An extensible type that is an extension of another type. A type that is declared
39 with the EXTENDS attribute.
- 40 **extensible type** (4.5.7) : A type from which new types may be derived using the EXTENDS attribute.

- 1 A nonsequence type that does not have the BIND attribute.
- 2 **extension type** (4.5.7) : A base type is an extension type of itself only. An extended type is an
3 extension type of itself and of all types for which its parent type is an extension.
- 4 **extent** (2.4.5) : The size of one dimension of an array.
- 5 **external file** (9.2) : A sequence of records that exists in a medium external to the program.
- 6 **external linkage** : The characteristic describing that a C entity is global to the program; defined in
7 subclause 6.2.2 of the C International Standard.
- 8 **external procedure** (2.2.4.1) : A procedure that is defined by an external subprogram or by a means
9 other than Fortran.
- 10 **external subprogram** (2.2) : A subprogram that is not in a main program, module, or another
11 subprogram. Note that a module is not called a subprogram. Note that in FORTRAN 77, a block data
12 program unit is called a subprogram.
- 13 **external unit** (9.4) : A mechanism that is used to refer to an external file. It is identified by a
14 nonnegative integer.
- 15 **file** (9) : An internal file or an external file.
- 16 **file storage unit** (9.2.4) : The unit of storage for an unformatted or stream file.
- 17 **final subroutine** (4.5.6) : A subroutine that is called automatically by the processor during finalization.
- 18 **finalizable** (4.5.6) : A type that has final subroutines, or that has a finalizable component. An object
19 of finalizable type.
- 20 **finalization** (4.5.6.2) : The process of calling user-defined final subroutines immediately before destroy-
21 ing an object.
- 22 **function** (2.2.4) : A procedure that is invoked in an expression and computes a value which is then
23 used in evaluating the expression.
- 24 **function result** (12.6.2.2) : The data object that returns the value of a function.
- 25 **function subprogram** (12.6.2.2) : A sequence of statements beginning with a FUNCTION statement
26 that is not in an interface block and ending with the corresponding END statement.
- 27 **generic identifier** (12.4.3.2) : A lexical token that appears in an INTERFACE statement and is
28 associated with all the procedures in the interface block or that appears in a GENERIC statement and
29 is associated with the specific type-bound procedures.
- 30 **generic interface** (4.5.2, 12.4.3.2) : An interface specified by a generic procedure binding or a generic
31 interface block.
- 32 **generic interface block** (12.4.3.2) : An interface block with a generic specification.
- 33 **global entity** (16.2) : An entity with an identifier whose scope is a program.
- 34 **host** (2.2.1) : Host scoping unit.
- 35 **host association** (16.5.1.4) : The process by which a contained scoping unit accesses entities of its
36 host.
- 37 **host scoping unit** (2.2.1) : A scoping unit that immediately surrounds another scoping unit.

- 1 **image** (2.3.2) : A Fortran program executes as if it were replicated a number of times, the number of
2 replications remaining fixed during execution of the program. Each copy is called an **image** and each
3 image executes asynchronously.
- 4 **image index** (2.3.2) : An integer value that identifies an image.
- 5 **implicit interface** (12.4.2) : For a procedure referenced in a scoping unit, the property of not having
6 an explicit interface. A statement function always has an implicit interface
- 7 **inherit** (4.5.7) : To acquire from a parent. Type parameters, components, or procedure bindings of an
8 extended type that are automatically acquired from its parent type without explicit declaration in the
9 extended type are said to be inherited.
- 10 **inheritance association** (4.5.7.2, 16.5.4) : The relationship between the inherited components and the
11 parent component in an extended type.
- 12 **inquiry function** (13.1) : A function that is either intrinsic or is defined in an intrinsic module and
13 whose result depends on properties of one or more of its arguments instead of their values.
- 14 **instance of a subprogram** (12.6.2.4) : The copy of a subprogram that is created when a procedure
15 defined by the subprogram is invoked.
- 16 **intent** (5.3.9) : An attribute of a dummy data object that indicates whether it is used to transfer data
17 into the procedure, out of the procedure, or both.
- 18 **interface block** (12.4.3.2) : A sequence of statements from an INTERFACE statement to the corre-
19 sponding END INTERFACE statement.
- 20 **interface body** (12.4.3.2) : A sequence of statements in an interface block from a FUNCTION or
21 SUBROUTINE statement to the corresponding END statement.
- 22 **internal file** (9.3) : A character variable that is used to transfer and convert data from internal storage
23 to internal storage.
- 24 **internal procedure** (2.2.4.3) : A procedure that is defined by an internal subprogram.
- 25 **internal subprogram** (2.2) : A subprogram in a main program or another subprogram.
- 26 **interoperable** (15.3) : The property of a Fortran entity that ensures that an equivalent entity may be
27 defined by means of C.
- 28 **intrinsic** (2.5.7) : An adjective that may be applied to types, operators, assignment statements, pro-
29 cedures, and modules. Intrinsic types, operators, and assignment statements are defined in this part of
30 ISO/IEC 1539 and may be used in any scoping unit without further definition or specification. Intrinsic
31 procedures are defined in this part of ISO/IEC 1539 or provided by a processor, and may be used in
32 a scoping unit without further definition or specification. Intrinsic modules are defined in this part of
33 ISO/IEC 1539 or provided by a processor, and may be accessed by use association; procedures and types
34 defined in an intrinsic module are not themselves intrinsic.
- 35 Intrinsic procedures and modules that are not defined in this part of ISO/IEC 1539 are called nonstandard
36 intrinsic procedures and modules.
- 37 **invoke** (2.2.4) :
- 38 To call a subroutine by a CALL statement or by a defined assignment statement.
- 39 To call a function by a reference to it by name or operator during the evaluation of an expression.
- To call a final subroutine by finalization.

- 1 **keyword** (2.5.2) : A word that is part of the syntax of a statement or a name that is used to identify
2 an item in a list.
- 3 **kind type parameter** (2.4.1.1, 4.4.2, 4.4.3, 4.4.4, 4.4.5, 4.4.6, 4.5.3) : A parameter whose values label
4 the available kinds of an intrinsic type, or a derived-type parameter that is declared to have the KIND
5 attribute.
- 6 **label** : See binding label or statement label.
- 7 **length of a character string** (4.4.5) : The number of characters in the character string.
- 8 **lexical token** (3.2) : A sequence of one or more characters with a specified interpretation.
- 9 **line** (3.3) : A sequence of 0 to 132 characters, which may contain Fortran statements, a comment, or
10 an INCLUDE line.
- 11 **linkage association** (16.5.1.5) : The association between interoperable Fortran entities and their C
12 counterparts.
- 13 **literal constant** (2.4.3.1.2, 4.4) : A constant without a name. Note that in FORTRAN 77, this was
14 called simply a constant.
- 15 **local entity** (16.3) : An entity identified by a lexical token whose scope is a scoping unit.
- 16 **local variable** (2.4.3.1.1) : A variable local to a particular scoping unit; not imported through use or
17 host association, not a dummy argument, and not a variable in common.
- 18 **main program** (2.3.6, 11.1) : A Fortran main program or a replacement defined by means other than
19 Fortran.
- 20 **many-one array section** (6.2.2.2.2) : An array section with a vector subscript having two or more
21 elements with the same value.
- 22 **module** (2.2.5, 11.2) : A program unit that contains or accesses definitions to be accessed by other
23 program units.
- 24 **module procedure** (2.2.4.2) : A procedure that is defined by a module subprogram.

J3 internal note

Unresolved Technical Issue 5003

The following glossary entry is just plain wrong. A Module Procedure Interface Block declares a Module Procedure Interface, but not a Separate Module Procedure's Interface unless the Separate Module Procedure is defined by an Separate Module Subprogram (sic).

This raises the question of whether the whole definition/declaration stuff in C12 is consistent and correct. It looks like it is probably inconsistent, since one cannot know when translating the module whether the MPIB should be treated as declaring the SMP's interface or not. When C12 has been fixed, the glossary entry can be reconsidered – and deleted if it is not useful.

- 25 **module procedure interface** (12.4.3.2) : The interface for a separate module procedure.
- 26 **module subprogram** (2.2) : A subprogram that is in a module but is not an internal subprogram.
- 27 **name** (3.2.1) : A lexical token consisting of a letter followed by up to 62 alphanumeric characters
28 (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.
- 29 **name association** (16.5.1) : Argument association, use association, host association, linkage associa-
30 tion, or construct association.

- 1 **named** : Having a name. That is, in a phrase such as “named variable,” the word “named” signifies that
2 the variable name is not qualified by a subscript list, substring specification, and so on. For example,
3 if X is an array variable, the reference “X” is a named variable while the reference “X(1)” is an object
4 designator.
- 5 **named constant** (2.4.3.1.2) : A constant that has a name. Note that in FORTRAN 77, this was called
6 a symbolic constant.
- 7 **NaN** (14.7) : A Not-a-Number value of IEEE arithmetic. It represents an undefined value or a value
8 created by an invalid operation.
- 9 **nonexecutable statement** (2.3.3) : A statement used to configure the program environment in which
10 computational actions take place.
- 11 **numeric storage unit** (16.5.3.2) : The unit of storage for holding a scalar that is not a pointer and is
12 of type default real, default integer, or default logical.
- 13 **numeric type** (4.4) : Integer, real or complex type.
- 14 **object** (2.4.3.1) : Data object.
- 15 **object designator** (2.5.1) : A designator for a data object.
- 16 **obsolescent feature** (1.8) : A feature that is considered to have been redundant but that is still in
17 frequent use.
- 18 **operand** (2.5.8) : An expression that precedes or succeeds an operator.
- 19 **operation** (7.1.5) : A computation involving one or two operands.
- 20 **operator** (2.5.8) : A lexical token that specifies an operation.
- 21 **override** (4.5.2, 4.5.7) : When explicit initialization or default initialization overrides default initializa-
22 tion, it is as if only the overriding initialization were specified. If a procedure is bound to an extensible
23 type, it overrides the one that would have been inherited from the parent type.
- 24 **parent component** (4.5.7.2) : The component of an entity of extended type that corresponds to its
25 inherited portion.
- 26 **parent type** (4.5.7) : The extensible type from which an extended type is derived.
- 27 **passed-object dummy argument** (4.5.4.4) : The dummy argument of a type-bound procedure or
28 procedure pointer component that becomes associated with the object through which the procedure was
29 invoked.
- 30 **pointer** (2.4.7) : An entity that has the POINTER attribute.
- 31 **pointer assignment** (7.2.2) : The pointer association of a pointer with a target by the execution of a
32 pointer assignment statement or the execution of an assignment statement for a data object of derived
33 type having the pointer as a subobject.
- 34 **pointer assignment statement** (7.2.2) : A statement of the form “pointer-object => target”.
- 35 **pointer associated** (6.3, 7.2.2) : The relationship between a pointer and a target following a pointer
36 assignment or a valid execution of an ALLOCATE statement.
- 37 **pointer association** (16.5.2) : The process by which a pointer becomes pointer associated with a target.
- 38 **polymorphic** (4.3.1.3) : Able to be of differing types during program execution. An object declared

- 1 with the CLASS keyword is polymorphic.
- 2 **preconnected** (9.4.4) : A property describing a unit that is connected to an external file at the beginning
3 of execution of a program. Such a unit may be specified in input/output statements without an OPEN
4 statement being executed for that unit.
- 5 **procedure** (2.2.4, 12.2) : A computation that may be invoked during program execution. It may be a
6 function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure,
7 an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than
8 one procedure if it contains ENTRY statements.
- 9 **procedure designator** (2.5.1) : A designator for a procedure.
- 10 **procedure interface** (12.4) : The characteristics of a procedure, the name of the procedure, the name
11 of each dummy argument, and the generic identifiers (if any) by which it may be referenced.
- 12 **processor** (1.2) : The combination of a computing system and the mechanism by which programs are
13 transformed for use on that computing system.
- 14 **processor dependent** (1.5) : The designation given to a facility that is not completely specified by
15 this part of ISO/IEC 1539. Such a facility shall be provided by a processor, with methods or semantics
16 determined by the processor.
- 17 **program** (2.2.2) : A set of program units that includes exactly one main program.
- 18 **program unit** (2.2.1) : The fundamental component of a program. A sequence of statements, comments,
19 and INCLUDE lines. It may be a main program, a module, an external subprogram, or a block data
20 program unit.
- 21 **prototype** : The C analog of a function interface body; defined in 6.7.5.3 of the C International
22 Standard.
- 23 **pure procedure** (12.7) : A procedure that is a pure intrinsic procedure (13.1), is defined by a pure
24 subprogram, or is a statement function that references only pure functions.
- 25 **rank** (2.4.4, 2.4.5) : The number of dimensions of an array. Zero for a scalar.
- 26 **record** (9.1) : A sequence of values or characters that is treated as a whole within a file.
- 27 **reference** (2.5.6) : The appearance of an object designator in a context requiring the value at that
28 point during execution, the appearance of a procedure designator, its operator symbol, or a defined
29 assignment statement in a context requiring execution of the procedure at that point, or the appearance
30 of a module name in a USE statement. Neither the act of defining a variable nor the appearance of the
31 name of a procedure as an actual argument is regarded as a reference.
- 32 **result variable** (2.2.4, 12.6.2.2) : The variable that returns the value of a function.
- 33 **rounding mode** (14.3, 10.7.2.3.7) : The method used to choose the result of an operation that cannot
34 be represented exactly. In IEEE arithmetic, there are four modes; nearest, towards zero, up (towards
35 ∞), and down (towards $-\infty$). In addition, for input/output the two additional modes COMPATIBLE
36 and PROCESSOR_DEFINED are provided.
- 37 **scalar** (2.4.4) :
38 Not being an array.
- 39 **scope** (16) : That part of a program within which a lexical token has a single interpretation. It may be
40 a program, a scoping unit, a construct, a single statement, or a part of a statement.

- 1 **scoping unit** (2.2.1) : One of the following:
- 2 (1) A program unit or subprogram, excluding any scoping units in it,
 - 3 (2) A derived-type definition, or
 - 4 (3) An interface body, excluding any scoping units in it.
- 5 **section subscript** (6.2.2) : A subscript, vector subscript, or subscript triplet in an array section selector.
- 6 **selector** (6.1.1, 6.1.2, 6.1.4, 8.1.5, 8.1.3) : A syntactic mechanism for designating
- 7 (1) a subobject,
 - 8 (2) the set of values for which a CASE block is executed,
 - 9 (3) the object whose type determines which branch of a SELECT TYPE construct is executed,
 - 10 or
 - 11 (4) the object that is associated with the *associate-name* in an ASSOCIATE construct.
- 12 **shape** (2.4.5) : The rank and extents of an array. The shape may be represented by the rank-one array
- 13 whose elements are the extents in each dimension.
- 14 **size** (2.4.5) : The total number of elements of an array.
- 15 **specification expression** (7.1.11) : An expression with limitations that make it suitable for use in
- 16 specifications such as length type parameters or array bounds.
- 17 **specification function** (7.1.11) : A nonintrinsic function that may be used in a specification expression.
- 18 **standard-conforming program** (1.5) : A program that uses only those forms and relationships de-
- 19 scribed in this part of ISO/IEC 1539, and that has an interpretation according to this part of ISO/IEC
- 20 1539.
- 21 **statement** (3.3) : A sequence of lexical tokens. It usually consists of a single line, but a statement may
- 22 be continued from one line to another and the semicolon symbol may be used to separate statements
- 23 within a line.
- 24 **statement entity** (16) : An entity identified by a lexical token whose scope is a single statement or
- 25 part of a statement.
- 26 **statement function** (12.6.4) : A procedure specified by a single statement that is similar in form to an assignment
- 27 statement.
- 28 **statement label** (3.2.4) : A lexical token consisting of up to five digits that precedes a statement and
- 29 may be used to refer to the statement.
- 30 **storage association** (16.5.3) : The relationship between two storage sequences if a storage unit of one
- 31 is the same as a storage unit of the other.
- 32 **storage sequence** (16.5.3.2) : A sequence of contiguous storage units.
- 33 **storage unit** (16.5.3.2) : A character storage unit, a numeric storage unit, a file storage unit, or an
- 34 unspecified storage unit.
- 35 **stride** (6.2.2.2.1) : The increment specified in a subscript triplet.
- 36 **struct** : The C analog of a sequence derived type; defined in 6.2.5 of the C International Standard.
- 37 **structure** (2.4.1.2) : A scalar data object of derived type.
- 38 **structure component** (6.1.2) : A part of an object of derived type. It may be referenced by an object
- designator.

- 1
- 2 **structure constructor** (4.5.10) : A syntactic mechanism for constructing a value of derived type.
- 3 **subcomponent** (6.1.2) : A subcomponent of an object of derived type is a component of that object
4 or of a subobject of that object.
- 5 **submodule** (2.2.6, 11.2.3) : A program unit that extends a module or another submodule.
- 6 **subobject** (2.4.3.1) : A portion of a data object that may be referenced or defined independently of
7 other portions. It may be an array element, an array section, a structure component, a substring, or the
8 real or imaginary part of a complex object.
- 9 **subprogram** (2.2.1) : A function subprogram or a subroutine subprogram. Note that in FORTRAN 77,
10 a block data program unit was called a subprogram.
- 11 **subroutine** (2.2.4) : A procedure that is invoked by a CALL statement or by a defined assignment
12 statement.
- 13 **subroutine subprogram** (12.6.2.3) : A sequence of statements beginning with a SUBROUTINE state-
14 ment that is not in an interface block and ending with the corresponding END statement.
- 15 **subscript** (6.2.2) : One of the list of scalar integer expressions in an array element selector. Note that
16 in FORTRAN 77, the whole list was called the subscript.
- 17 **subscript triplet** (6.2.2) : An item in the list of an array section selector that contains a colon and
18 specifies a regular sequence of integer values.
- 19 **substring** (6.1.1) : A contiguous portion of a scalar character string. Note that an array section can
20 include a substring selector; the result is called an array section and not a substring.
- 21 **target** (2.4.7, 5.3.16, 6.3.1.2) : A data entity that has the TARGET attribute, or an entity that is
22 associated with a pointer.
- 23 **team** (2.3.2) : A set of images formed by invoking the intrinsic collective subroutine FORM_TEAM
24 (13.7.69).
- 25 **team synchronization** (8.5.3) : Synchronization of the images in a team.
- 26 **transformational function** (13.1) : A function that is either intrinsic or is defined in an intrinsic
27 module and that is neither an elemental function nor an inquiry function.
- 28 **type** (2.4.1) : A named category of data that is characterized by a set of values, together with a way to
29 denote these values and a collection of operations that interpret and manipulate the values. The set of
30 data values depends on the values of the type parameters.
- 31 **type-bound procedure** (4.5.5) : A procedure binding in a type definition. The procedure may be
32 referenced by the *binding-name* via any object of that dynamic type, as a defined operator, by defined
33 assignment, or as part of the finalization process.
- 34 **type compatible** (4.3.1.3) : All entities are type compatible with other entities of the same type.
35 Unlimited polymorphic entities are type compatible with all entities; other polymorphic entities are type
36 compatible with entities whose dynamic type is an extension type of the polymorphic entity's declared
37 type.
- 38 **type declaration statement** (5.2) : An INTEGER, REAL, DOUBLE PRECISION, COMPLEX,
39 CHARACTER, LOGICAL, TYPE (*type-name*), or CLASS (*type-name*) statement.
- 40 **type parameter** (2.4.1.1) : A parameter of a data type. KIND and LEN are the type parameters of

- 1 intrinsic types. The type parameters of a derived type are defined in the derived-type definition.
- 2 **type parameter order** (4.5.3.2) : The ordering of the type parameters of a derived type that is used
3 for derived-type specifiers.
- 4 **ultimate component** (4.5) : For a structure, a component that is of intrinsic type, has the ALLOCAT-
5 ABLE attribute, or has the POINTER attribute, or an ultimate component of a derived-type component
6 that does not have the POINTER attribute or the ALLOCATABLE attribute.
- 7 **undefined** (2.5.5) : For a data object, the property of not having a determinate value.
- 8 **unsigned** : A qualifier of a C numeric type indicating that it is comprised only of nonnegative values;
9 defined in 6.2.5 of the C International Standard. There is nothing analogous in Fortran.
- 10 **unspecified storage unit** (16.5.3.2) : A unit of storage for holding a pointer or a scalar that is not a
11 pointer and is of type other than default integer, default character, default real, double precision real,
12 default logical, or default complex.
- 13 **use association** (16.5.1.3) : The association of names in different scoping units specified by a USE
14 statement.
- 15 **variable** (2.4.3.1.1) : A data object whose value can be defined and redefined during the execution of
16 a program. It may be a named data object, an array element, an array section, a structure component,
17 or a substring. Note that in FORTRAN 77, a variable was always scalar and named.
- 18 **vector subscript** (6.2.2.2.2) : A section subscript that is an integer expression of rank one.
- 19 **void** : A C type comprising an empty set of values; defined in 6.2.5 of the C International Standard.
20 There is nothing analogous in Fortran.
- 21 **whole array** (6.2.1) : A named array.

Annex B

(Informative)

Decremental features

B.1 Deleted features

The deleted features are those features of Fortran 90 that were redundant and considered largely unused.

The following Fortran 90 features are not required by Fortran 95, Fortran 2003, or this part of ISO/IEC 1539.

- (1) Real and double precision DO variables.
The ability present in FORTRAN 77, and for consistency also in Fortran 90, for a DO variable to be of type real or double precision in addition to type integer, 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 for consistency also in 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, present in FORTRAN 66, FORTRAN 77 and for consistency also in 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, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, present in FORTRAN 77 and Fortran 90, has been deleted. A similar result can be achieved by using other control constructs instead of the assigned GOTO 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 for consistency also in 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.

The following is a list of the previous editions of the Fortran International Standard, along with their informal names.

- ISO R 1539-1972, FORTRAN 66;
- ISO 1539-1980, FORTRAN 77;
- ISO/IEC 1539:1991, Fortran 90;
- ISO/IEC 1539-1:1997, Fortran 95.

See ISO/IEC 1539:1991 for detailed rules of how these deleted features work.

1 B.2 Obsolescent features

2 The obsolescent features are those features of Fortran 90 that were redundant and for which better
3 methods were available in Fortran 90. Subclause 1.8.3 describes the nature of the obsolescent features.

4 The obsolescent features in this part of ISO/IEC 1539 are the following.

- 5 (1) Arithmetic IF — use the IF statement or IF construct (8.1.8).
- 6 (2) Shared DO termination and termination on a statement other than END DO or CON-
7 TINUE — use an END DO or a CONTINUE statement for each DO statement.
- 8 (3) Alternate return — see B.2.1.
- 9 (4) Computed GO TO statement — see B.2.2.
- 10 (5) Statement functions — see B.2.3.
- 11 (6) DATA statements amongst executable statements — see B.2.4.
- 12 (7) Assumed length character functions — see B.2.5.
- 13 (8) Fixed form source — see B.2.6.
- 14 (9) CHARACTER* form of CHARACTER declaration — see B.2.7.

15 B.2.1 Alternate return

16 An alternate return introduces labels into an argument list to allow the called procedure to direct the
17 execution of the caller upon return. The same effect can be achieved with a return code that is used
18 in a CASE construct on return. This avoids an irregularity in the syntax and semantics of argument
19 association. For example,

20 CALL SUBR_NAME (X, Y, Z, *100, *200, *300)

21 may be replaced by

```
22 CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
23 SELECT CASE (RETURN_CODE)
24 CASE (1)
25 ...
26 CASE (2)
27 ...
28 CASE (3)
29 ...
30 CASE DEFAULT
31 ...
32 END SELECT
```

33 B.2.2 Computed GO TO statement

34 The computed GO TO has been superseded by the CASE construct, which is a generalized, easier to
35 use and more efficient means of expressing the same computation.

36 B.2.3 Statement functions

37 Statement functions are subject to a number of nonintuitive restrictions and are a potential source of
38 error because their syntax is easily confused with that of an assignment statement.

39 The internal function is a more generalized form of the statement function and completely supersedes
40 it.

1 **B.2.4 DATA statements among executables**

2 The statement ordering rules of FORTRAN 66, and hence of FORTRAN 77 and Fortran 90 for compatibility,
3 allowed DATA statements to appear anywhere in a program unit after the specification statements. The
4 ability to position DATA statements amongst executable statements is very rarely used, is unnecessary
5 and is a potential source of error.

6 **B.2.5 Assumed character length functions**

7 Assumed character length for functions is an irregularity in the language in that elsewhere in Fortran
8 the philosophy is that the attributes of a function result depend only on the actual arguments of the
9 invocation and on any data accessible by the function through host or use association. Some uses of this
10 facility can be replaced with an automatic character length function, where the length of the function
11 result is declared in a specification expression. Other uses can be replaced by the use of a subroutine
12 whose arguments correspond to the function result and the function arguments.

13 Note that dummy arguments of a function may be assumed character length.

14 **B.2.6 Fixed form source**

15 Fixed form source was designed when the principal machine-readable input medium for new programs
16 was punched cards. Now that new and amended programs are generally entered via keyboards with
17 screen displays, it is an unnecessary overhead, and is potentially error-prone, to have to locate positions
18 6, 7, or 72 on a line. Free form source was designed expressly for this more modern technology.

19 It is a simple matter for a software tool to convert from fixed to free form source.

20 **B.2.7 CHARACTER* form of CHARACTER declaration**

21 Fortran 90 had two different forms of specifying the length selector in CHARACTER declarations. The
22 older form (CHARACTER*char-length) is redundant.

Annex C

(Informative)

Extended notes

C.1 Clause 4 notes

C.1.1 Selection of the approximation methods (4.4.3)

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.

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 inquiry function SELECTED_REAL_KIND. This function, given integer arguments P and R specifying minimum requirements for decimal precision and decimal exponent range, respectively, returns the kind type parameter value of the approximation method that has at least P decimal digits of precision and at least a range for positive numbers of 10^{-R} to 10^R . In the above specification statement, the 8 may 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 may have flexibility in assigning such values and may add new kinds without changing previously assigned kind values.

As kind values have no portable meaning, a good practice is to use them in programs only through named constants as described above (for example, SINGLE, IEEE_SINGLE, DOUBLE, and QUAD), rather than using the kind values directly.

C.1.2 Type extension and component accessibility (4.5.2.2, 4.5.4)

The default accessibility of an extended type may be specified in the type definition. The accessibility of its components may be specified individually.

```
module types
  type base_type
    private          !-- Sets default accessibility
```

```

1      integer :: i          !-- a private component
2      integer, private :: j !-- another private component
3      integer, public :: k  !-- a public component
4  end type base_type
5
6  type, extends(base_type) :: my_type
7      private              !-- Sets default for components declared in my_type
8      integer :: l          !-- A private component.
9      integer, public :: m !-- A public component.
10 end type my_type
11
12 end module types
13
14 subroutine sub
15     use types
16     type (my_type) :: x
17
18     ....
19
20     call another_sub( &
21         x%base_type, & !-- ok because base_type is a public subobject of x
22         x%base_type%k, & !-- ok because x%base_type is ok and has k as a
23                         !-- public component.
24         x%k,          & !-- ok because it is shorthand for x%base_type%k
25         x%base_type%i, & !-- Invalid because i is private.
26         x%i)          !-- Invalid because it is shorthand for x%base_type%i
27 end subroutine sub

```

28 C.1.3 Abstract types

29 The following defines an object that can be displayed in an X window:

```

30     TYPE, ABSTRACT :: DRAWABLE_OBJECT
31         REAL, DIMENSION(3) :: RGB_COLOR = (/1.0,1.0,1.0/) ! White
32         REAL, DIMENSION(2) :: POSITION = (/0.0,0.0/) ! Centroid
33     CONTAINS
34         PROCEDURE(RENDER_X), PASS(OBJECT), DEFERRED :: RENDER
35     END TYPE DRAWABLE_OBJECT
36
37     ABSTRACT INTERFACE
38         SUBROUTINE RENDER_X(OBJECT, WINDOW)
39             CLASS(DRAWABLE_OBJECT), INTENT(IN) :: OBJECT
40             CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW
41         END SUBROUTINE RENDER_X
42     END INTERFACE

```

1 We can declare a nonabstract type by extending the abstract type:

```

2      TYPE, EXTENDS(DRAWABLE_OBJECT) :: DRAWABLE_TRIANGLE ! Not ABSTRACT
3      REAL, DIMENSION(2,3) :: VERTICES ! In relation to centroid
4      CONTAINS
5      PROCEDURE, PASS(OBJECT) :: RENDER=>RENDER_TRIANGLE_X
6      END TYPE DRAWABLE_TRIANGLE

```

7 The actual drawing procedure will draw a triangle in WINDOW with vertices at x coordinates
8 OBJECT%POSITION(1)+OBJECT%VERTICES(1,:) and y coordinates
9 OBJECT%POSITION(2)+OBJECT%VERTICES(2,:):

```

10     SUBROUTINE RENDER_TRIANGLE_X(OBJECT, WINDOW)
11     CLASS(DRAWABLE_TRIANGLE), INTENT(IN) :: OBJECT
12     CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW
13     ...
14     END SUBROUTINE RENDER_TRIANGLE_X

```

15 C.1.4 Pointers (4.5.2)

16 Pointers are names that can change dynamically their association with a target object. In a sense, a
17 normal variable is a name with a fixed association with a particular object. A normal variable name
18 refers to the same storage space throughout the lifetime of the variable. A pointer name may refer
19 to different storage space, or even no storage space, at different times. A variable may be considered
20 to be a descriptor for space to hold values of the appropriate type, type parameters, and array rank
21 such that the values stored in the descriptor are fixed when the variable is created. A pointer also may
22 be considered to be a descriptor, but one whose values may be changed dynamically so as to describe
23 different pieces of storage. When a pointer is declared, space to hold the descriptor is created, but the
24 space for the target object is not created.

25 A derived type may have one or more components that are defined to be pointers. It may have a
26 component that is a pointer to an object of the same derived type. This “recursive” data definition
27 allows dynamic data structures such as linked lists, trees, and graphs to be constructed. For example:

```

28 TYPE NODE          ! Define a ''recursive'' type
29     INTEGER :: VALUE = 0
30     TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
31 END TYPE NODE
32
33 TYPE (NODE), TARGET :: HEAD          ! Automatically initialized
34 TYPE (NODE), POINTER :: CURRENT, TEMP ! Declare pointers
35 INTEGER :: IOEM, K
36
37 CURRENT => HEAD                      ! CURRENT points to head of list
38
39 DO
40     READ (*, *, IOSTAT = IOEM) K ! Read next value, if any

```

```

1   IF (IOEM /= 0) EXIT
2   ALLOCATE (TEMP)           ! Create new cell each iteration
3   TEMP % VALUE = K         ! Assign value to cell
4   CURRENT % NEXT_NODE => TEMP ! Attach new cell to list
5   CURRENT => TEMP           ! CURRENT points to new end of list
6 END DO

```

7 A list is now constructed and the last linked cell contains a disassociated pointer. A loop can be used
8 to “walk through” the list.

```

9 CURRENT => HEAD
10 DO
11   IF (.NOT. ASSOCIATED (CURRENT % NEXT_NODE)) EXIT
12   CURRENT => CURRENT % NEXT_NODE
13   WRITE (*, *) CURRENT % VALUE
14 END DO

```

15 C.1.5 Structure constructors and generic names

16 A generic name may be the same as a type name. This can be used to emulate user-defined structure
17 constructors for that type, even if the type has private components. For example:

```

18 MODULE mytype_module
19   TYPE mytype
20     PRIVATE
21     COMPLEX value
22     LOGICAL exact
23   END TYPE
24   INTERFACE mytype
25     MODULE PROCEDURE int_to_mytype
26   END INTERFACE
27   ! Operator definitions etc.
28   ...
29 CONTAINS
30   TYPE(mytype) FUNCTION int_to_mytype(i)
31     INTEGER, INTENT(IN) :: i
32     int_to_mytype%value = i
33     int_to_mytype%exact = .TRUE.
34   END FUNCTION
35   ! Procedures to support operators etc.
36   ...
37 END
38
39 PROGRAM example
40   USE mytype_module

```



```

1  TYPE(mytype) x
2  x = mytype(17)
3  END

```

4 The type name may still be used as a generic name if the type has type parameters. For example:

```

5  MODULE m
6  TYPE t(kind)
7      INTEGER, KIND :: kind
8      COMPLEX(kind) value
9  END TYPE
10  INTEGER, PARAMETER :: single = KIND(0.0), double = KIND(0d0)
11  INTERFACE t
12      MODULE PROCEDURE real_to_t1, dble_to_t2, int_to_t1, int_to_t2
13  END INTERFACE
14  ...
15  CONTAINS
16  TYPE(t(single)) FUNCTION real_to_t1(x)
17      REAL(single) x
18      real_to_t1%value = x
19  END FUNCTION
20  TYPE(t(double)) FUNCTION dble_to_t2(x)
21      REAL(double) x
22      dble_to_t2%value = x
23  END FUNCTION
24  TYPE(t(single)) FUNCTION int_to_t1(x,mold)
25      INTEGER x
26      TYPE(t(single)) mold
27      int_to_t1%value = x
28  END FUNCTION
29  TYPE(t(double)) FUNCTION int_to_t2(x,mold)
30      INTEGER x
31      TYPE(t(double)) mold
32      int_to_t2%value = x
33  END FUNCTION
34  ...
35  END
36
37  PROGRAM example
38      USE m
39      TYPE(t(single)) x
40      TYPE(t(double)) y
41      x = t(1.5)                ! References real_to_t1
42      x = t(17,mold=x)          ! References int_to_t1
43      y = t(1.5d0)              ! References dble_to_t2

```

```

1
2   y = t(42,mold=y)           ! References int_to_t2
3   y = t(kind(0d0)) ((0,1))  ! Uses the structure constructor for type t
4 END

```

5 C.1.6 Generic type-bound procedures

6 Example of a derived type with generic type-bound procedures:

7 The only difference between this example and the same thing rewritten to use generic interface blocks
8 is that with type-bound procedures,

```

9     USE(rational_numbers),ONLY :: rational

```

10 does not block the type-bound procedures; the user still gets access to the defined assignment and
11 extended operations.

```

12 MODULE rational_numbers
13   IMPLICIT NONE
14   PRIVATE
15   TYPE,PUBLIC :: rational
16     PRIVATE
17     INTEGER n,d
18   CONTAINS
19     ! ordinary type-bound procedure
20     PROCEDURE :: real => rat_to_real
21     ! specific type-bound procedures for generic support
22     PROCEDURE,PRIVATE :: rat_asgn_i
23     PROCEDURE,PRIVATE :: rat_plus_rat
24     PROCEDURE,PRIVATE :: rat_plus_i
25     PROCEDURE,PRIVATE,PASS(b) :: i_plus_rat
26     ! generic type-bound procedures
27     GENERIC :: ASSIGNMENT(=) => rat_asgn_i
28     GENERIC :: OPERATOR(+) => rat_plus_rat, rat_plus_i, i_plus_rat
29   END TYPE
30 CONTAINS
31   ELEMENTAL REAL FUNCTION rat_to_real(this) RESULT(r)
32     CLASS(rational),INTENT(IN) :: this
33     r = REAL(this%n)/this%d
34   END FUNCTION
35   ELEMENTAL SUBROUTINE rat_asgn_i(a,b)
36     CLASS(rational),INTENT(OUT) :: a
37     INTEGER,INTENT(IN) :: b
38     a%n = b
39     a%d = 1
40   END SUBROUTINE

```

```

1  ELEMENTAL TYPE(rational) FUNCTION rat_plus_i(a,b) RESULT(r)
2      CLASS(rational),INTENT(IN) :: a
3      INTEGER,INTENT(IN) :: b
4      r%n = a%n + b*a%d
5      r%d = a%d
6  END FUNCTION
7  ELEMENTAL TYPE(rational) FUNCTION i_plus_rat(a,b) RESULT(r)
8      INTEGER,INTENT(IN) :: a
9      CLASS(rational),INTENT(IN) :: b
10     r%n = b%n + a*b%d
11     r%d = b%d
12 END FUNCTION
13 ELEMENTAL TYPE(rational) FUNCTION rat_plus_rat(a,b) RESULT(r)
14     CLASS(rational),INTENT(IN) :: a,b
15     r%n = a%n*b%d + b%n*a%d
16     r%d = a%d*b%d
17 END FUNCTION
18 END

```

19 C.1.7 Final subroutines (4.5.6, 4.5.6.2, 4.5.6.3, 4.5.6.4)

20 Example of a parameterized derived type with final subroutines:

```

21 MODULE m
22     TYPE t(k)
23         INTEGER, KIND :: k
24         REAL(k),POINTER :: vector(:) => NULL()
25     CONTAINS
26         FINAL :: finalize_t1s, finalize_t1v, finalize_t2e
27     END TYPE
28 CONTAINS
29     SUBROUTINE finalize_t1s(x)
30         TYPE(t(KIND(0.0))) x
31         IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
32     END SUBROUTINE
33     SUBROUTINE finalize_t1v(x)
34         TYPE(t(KIND(0.0))) x(:)
35         DO i=LBOUND(x,1),UBOUND(x,1)
36             IF (ASSOCIATED(x(i)%vector)) DEALLOCATE(x(i)%vector)
37         END DO
38     END SUBROUTINE
39     ELEMENTAL SUBROUTINE finalize_t2e(x)
40         TYPE(t(KIND(0.0d0))),INTENT(INOUT) :: x
41         IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
42     END SUBROUTINE

```

```

1  END MODULE
2
3  SUBROUTINE example(n)
4      USE m
5      TYPE(t(KIND(0.0))) a,b(10),c(n,2)
6      TYPE(t(KIND(0.0d0))) d(n,n)
7      ...
8      ! Returning from this subroutine will effectively do
9      !     CALL finalize_t1s(a)
10     !     CALL finalize_t1v(b)
11     !     CALL finalize_t2e(d)
12     ! No final subroutine will be called for variable C because the user
13     ! omitted to define a suitable specific procedure for it.
14 END SUBROUTINE

```

15 Example of extended types with final subroutines:

```

16 MODULE m
17     TYPE t1
18         REAL a,b
19     END TYPE
20     TYPE,EXTENDS(t1) :: t2
21         REAL,POINTER :: c(:),d(:)
22     CONTAINS
23         FINAL :: t2f
24     END TYPE
25     TYPE,EXTENDS(t2) :: t3
26         REAL,POINTER :: e
27     CONTAINS
28         FINAL :: t3f
29     END TYPE
30     ...
31 CONTAINS
32     SUBROUTINE t2f(x) ! Finalizer for TYPE(t2)'s extra components
33         TYPE(t2) :: x
34         IF (ASSOCIATED(x%c)) DEALLOCATE(x%c)
35         IF (ASSOCIATED(x%d)) DEALLOCATE(x%d)
36     END SUBROUTINE
37     SUBROUTINE t3f(y) ! Finalizer for TYPE(t3)'s extra components
38         TYPE(t3) :: y
39         IF (ASSOCIATED(y%e)) DEALLOCATE(y%e)
40     END SUBROUTINE
41 END MODULE
42
43 SUBROUTINE example

```

```

1
2   USE m
3   TYPE(t1) x1
4   TYPE(t2) x2
5   TYPE(t3) x3
6   ...
7   ! Returning from this subroutine will effectively do
8   !   ! Nothing to x1; it is not finalizable
9   !   CALL t2f(x2)
10  !   CALL t3f(x3)
11  !   CALL t2f(x3%t2)
12 END SUBROUTINE

```

13 C.2 Clause 5 notes

14 C.2.1 The POINTER attribute (5.3.13)

15 The POINTER attribute shall be specified to declare a pointer. The type, type parameters, and rank,
 16 which may be specified in the same statement or with one or more attribute specification statements,
 17 determine the characteristics of the target objects that may be associated with the pointers declared
 18 in the statement. An obvious model for interpreting declarations of pointers is that such declarations
 19 create for each name a descriptor. Such a descriptor includes all the data necessary to describe fully
 20 and locate in memory an object and all subobjects of the type, type parameters, and rank specified.
 21 The descriptor is created empty; it does not contain values describing how to access an actual memory
 22 space. These descriptor values will be filled in when the pointer is associated with actual target space.

23 The following example illustrates the use of pointers in an iterative algorithm:

```

24 PROGRAM DYNAM_ITER
25   REAL, DIMENSION (:, :), POINTER :: A, B, SWAP ! Declare pointers
26   ...
27   READ (*, *) N, M
28   ALLOCATE (A (N, M), B (N, M)) ! Allocate target arrays
29   ! Read values into A
30   ...
31   ITER: DO
32     ...
33     ! Apply transformation of values in A to produce values in B
34     ...
35     IF (CONVERGED) EXIT ITER
36     ! Swap A and B
37     SWAP => A; A => B; B => SWAP
38   END DO ITER
39   ...
40 END PROGRAM DYNAM_ITER

```

C.2.2 The TARGET attribute (5.3.16)

1

2 The TARGET attribute shall be specified for any nonpointer object that may, during the execution of the
 3 program, become associated with a pointer. This attribute is defined primarily for optimization purposes.
 4 It allows the processor to assume that any nonpointer object not explicitly declared as a target may
 5 be referred to only by way of its original declared name. It also means that implicitly-declared objects
 6 shall not be used as pointer targets. This will allow a processor to perform optimizations that otherwise
 7 would not be possible in the presence of certain pointers.

8 The following example illustrates the use of the TARGET attribute in an iterative algorithm:

```

9 PROGRAM ITER
10   REAL, DIMENSION (1000, 1000), TARGET :: A, B
11   REAL, DIMENSION (:, :), POINTER      :: IN, OUT, SWAP
12   ...
13   ! Read values into A
14   ...
15   IN => A           ! Associate IN with target A
16   OUT => B          ! Associate OUT with target B
17   ...
18   ITER:DO
19     ...
20     ! Apply transformation of IN values to produce OUT
21     ...
22     IF (CONVERGED) EXIT ITER
23     ! Swap IN and OUT
24     SWAP => IN; IN => OUT; OUT => SWAP
25   END DO ITER
26   ...
27 END PROGRAM ITER

```

28 C.2.3 The VOLATILE attribute (5.3.18)

29 The following example shows the use of a variable with the VOLATILE attribute to communicate with
 30 an asynchronous process, in this case the operating system. The program detects a user keystroke on
 31 the terminal and reacts at a convenient point in its processing.

32 The VOLATILE attribute is necessary to prevent an optimizing compiler from storing the communication
 33 variable in a register or from doing flow analysis and deciding that the EXIT statement can never be
 34 executed.

```

35 SUBROUTINE TERMINATE_ITERATIONS
36
37   LOGICAL, VOLATILE :: USER_HIT_ANY_KEY
38
39   ! Have the OS start to look for a user keystroke and set the variable
40   ! "USER_HIT_ANY_KEY" to TRUE as soon as it detects a keystroke.
41   ! This pseudo call is operating system dependent.

```

```

1
2  CALL OS_BEGIN_DETECT_USER_KEYSTROKE( USER_HIT_ANY_KEY )
3
4  USER_HIT_ANY_KEY = .FALSE.      ! This will ignore any recent keystrokes
5
6  PRINT *, " Hit any key to terminate iterations!"
7
8  DO I = 1,100
9      ...                        ! Compute a value for R
10     PRINT *, I, R
11     IF (USER_HIT_ANY_KEY)      EXIT
12 ENDDO
13
14 ! Have the OS stop looking for user keystrokes
15 CALL OS_STOP_DETECT_USER_KEYSTROKE
16
17 END SUBROUTINE TERMINATE_ITERATIONS

```

18 C.3 Clause 6 notes

19 C.3.1 Structure components (6.1.2)

20 Components of a structure are referenced by writing the components of successive levels of the structure
 21 hierarchy until the desired component is described. For example,

```

22 TYPE ID_NUMBERS
23     INTEGER SSN
24     INTEGER EMPLOYEE_NUMBER
25 END TYPE ID_NUMBERS
26
27 TYPE PERSON_ID
28     CHARACTER (LEN=30) LAST_NAME
29     CHARACTER (LEN=1) MIDDLE_INITIAL
30     CHARACTER (LEN=30) FIRST_NAME
31     TYPE (ID_NUMBERS) NUMBER
32 END TYPE PERSON_ID
33
34 TYPE PERSON
35     INTEGER AGE
36     TYPE (PERSON_ID) ID
37 END TYPE PERSON
38
39 TYPE (PERSON) GEORGE, MARY
40
41 PRINT *, GEORGE % AGE          ! Print the AGE component

```

```

1
2 PRINT *, MARY % ID % LAST_NAME      ! Print LAST_NAME of MARY
3 PRINT *, MARY % ID % NUMBER % SSN ! Print SSN of MARY
4 PRINT *, GEORGE % ID % NUMBER      ! Print SSN and EMPLOYEE_NUMBER of GEORGE

```

5 A structure component may be a data object of intrinsic type as in the case of GEORGE % AGE or it
 6 may be of derived type as in the case of GEORGE % ID % NUMBER. The resultant component may
 7 be a scalar or an array of intrinsic or derived type.

```

8 TYPE LARGE
9     INTEGER ELT (10)
10    INTEGER VAL
11 END TYPE LARGE
12
13 TYPE (LARGE) A (5)          ! 5 element array, each of whose elements
14                             ! includes a 10 element array ELT and
15                             ! a scalar VAL.
16 PRINT *, A (1)              ! Prints 10 element array ELT and scalar VAL.
17 PRINT *, A (1) % ELT (3)    ! Prints scalar element 3
18                             ! of array element 1 of A.
19 PRINT *, A (2:4) % VAL      ! Prints scalar VAL for array elements
20                             ! 2 to 4 of A.

```

21 Components of an object of extensible type that are inherited from the parent type may be accessed as
 22 a whole by using the parent component name, or individually, either with or without qualifying them
 23 by the parent component name.

24 For example:

```

25 TYPE POINT                  ! A base type
26     REAL :: X, Y
27 END TYPE POINT
28 TYPE, EXTENDS(POINT) :: COLOR_POINT ! An extension of TYPE(POINT)
29     ! Components X and Y, and component name POINT, inherited from parent
30     INTEGER :: COLOR
31 END TYPE COLOR_POINT
32
33 TYPE(POINT) :: PV = POINT(1.0, 2.0)
34 TYPE(COLOR_POINT) :: CPV = COLOR_POINT(POINT=PV, COLOR=3)
35
36 PRINT *, CPV%POINT          ! Prints 1.0 and 2.0
37 PRINT *, CPV%POINT%X, CPV%POINT%Y ! And this does, too
38 PRINT *, CPV%X, CPV%Y       ! And this does, too

```

39 C.3.2 Allocation with dynamic type (6.3.1)

1 The following example illustrates the use of allocation with the value and dynamic type of the allocated
 2 object given by another object. The example copies a list of objects of any type. It copies the list
 3 starting at IN_LIST. After copying, each element of the list starting at LIST_COPY has a polymorphic
 4 component, ITEM, for which both the value and type are taken from the ITEM component of the
 5 corresponding element of the list starting at IN_LIST.

```

6  TYPE :: LIST ! A list of anything
7    TYPE(LIST), POINTER :: NEXT => NULL()
8    CLASS(*), ALLOCATABLE :: ITEM
9  END TYPE LIST
10 ...
11 TYPE(LIST), POINTER :: IN_LIST, LIST_COPY => NULL()
12 TYPE(LIST), POINTER :: IN_WALK, NEW_TAIL
13 ! Copy IN_LIST to LIST_COPY
14 IF (ASSOCIATED(IN_LIST)) THEN
15   IN_WALK => IN_LIST
16   ALLOCATE(LIST_COPY)
17   NEW_TAIL => LIST_COPY
18   DO
19     ALLOCATE(NEW_TAIL%ITEM, SOURCE=IN_WALK%ITEM)
20     IN_WALK => IN_WALK%NEXT
21     IF (.NOT. ASSOCIATED(IN_WALK)) EXIT
22     ALLOCATE(NEW_TAIL%NEXT)
23     NEW_TAIL => NEW_TAIL%NEXT
24   END DO
25 END IF

```

26 C.3.3 Pointer allocation and association

27 The effect of ALLOCATE, DEALLOCATE, NULLIFY, and pointer assignment is that they are inter-
 28 preted as changing the values in the descriptor that is the pointer. An ALLOCATE is assumed to create
 29 space for a suitable object and to “assign” to the pointer the values necessary to describe that space.
 30 A NULLIFY breaks the association of the pointer with the space. A DEALLOCATE breaks the asso-
 31 ciation and releases the space. Depending on the implementation, it could be seen as setting a flag in
 32 the pointer that indicates whether the values in the descriptor are valid, or it could clear the descriptor
 33 values to some (say zero) value indicative of the pointer not pointing to anything. A pointer assignment
 34 copies the values necessary to describe the space occupied by the target into the descriptor that is the
 35 pointer. Descriptors are copied, values of objects are not.

36 If PA and PB are both pointers and PB is associated with a target, then

```
37 PA => PB
```

38 results in PA being associated with the same target as PB. If PB was disassociated, then PA becomes
 39 disassociated.

40 This part of ISO/IEC 1539 is specified so that such associations are direct and independent. A subsequent
 41 statement

```
42 PB => D
```

1 or

2 ALLOCATE (PB)

3 has no effect on the association of PA with its target. A statement

4 DEALLOCATE (PB)

5 deallocates the space that is associated with both PA and PB. PB becomes disassociated, but there is
6 no requirement that the processor make it explicitly recognizable that PA no longer has a target. This
7 leaves PA as a “dangling pointer” to space that has been released. The program shall not use PA again
8 until it becomes associated via pointer assignment or an ALLOCATE statement.

9 DEALLOCATE may only be used to release space that was created by a previous ALLOCATE. Thus
10 the following is invalid:

11 REAL, TARGET :: T

12 REAL, POINTER :: P

13 ...

14 P = > T

15 DEALLOCATE (P) ! Not allowed: P's target was not allocated

16 The basic principle is that ALLOCATE, NULLIFY, and pointer assignment primarily affect the pointer
17 rather than the target. ALLOCATE creates a new target but, other than breaking its connection with
18 the specified pointer, it has no effect on the old target. Neither NULLIFY nor pointer assignment has
19 any effect on targets. A piece of memory that was allocated and associated with a pointer will become
20 inaccessible to a program if the pointer is nullified or associated with a different target and no other
21 pointer was associated with this piece of memory. Such pieces of memory may be reused by the processor
22 if this is expedient. However, whether such inaccessible memory is in fact reused is entirely processor
23 dependent.

24 C.4 Clause 7 notes

25 C.4.1 Character assignment

26 The FORTRAN 77 restriction that none of the character positions being defined in the character assign-
27 ment statement may be referenced in the expression has been removed (7.2.1.3).

28 C.4.2 Evaluation of function references

29 If more than one function reference appears in a statement, they may be executed in any order (subject to
30 a function result being evaluated after the evaluation of its arguments) and their values shall not depend
31 on the order of execution. This lack of dependence on order of evaluation permits parallel execution of
32 the function references (7.1.7).

33 C.4.3 Pointers in expressions

34 A pointer is basically considered to be like any other variable when it is used as a primary in an expression.
35 If a pointer is used as an operand to an operator that expects a value, the pointer will automatically
36 deliver the value stored in the space described by the pointer, that is, the value of the target object
37 associated with the pointer.

1 C.4.4 Pointers on the left side of an assignment

2 A pointer that appears on the left of an intrinsic assignment statement also is dereferenced and is taken
3 to be referring to the space that is its current target. Therefore, the assignment statement specifies the
4 normal copying of the value of the right-hand expression into this target space. All the normal rules of
5 intrinsic assignment hold; the type and type parameters of the expression and the pointer target shall
6 agree and the shapes shall be conformable.

7 For intrinsic assignment of derived types, nonpointer components are assigned and pointer components
8 are pointer assigned. Dereferencing is applied only to entire scalar objects, not selectively to pointer
9 subobjects.

10 For example, suppose a type such as

```
11 TYPE CELL
12     INTEGER :: VAL
13     TYPE (CELL), POINTER :: NEXT_CELL
14 END TYPE CELL
```

15 is defined and objects such as HEAD and CURRENT are declared using

```
16 TYPE (CELL), TARGET :: HEAD
17 TYPE (CELL), POINTER :: CURRENT
```

18 If a linked list has been created and attached to HEAD and the pointer CURRENT has been allocated
19 space, statements such as

```
20 CURRENT = HEAD
21 CURRENT = CURRENT % NEXT_CELL
```

22 cause the contents of the cells referenced on the right to be copied to the cell referred to by CURRENT.
23 In particular, the right-hand side of the second statement causes the pointer component in the cell,
24 CURRENT, to be selected. This pointer is dereferenced because it is in an expression context to produce
25 the target's integer value and a pointer to a cell that is in the target's NEXT_CELL component. The
26 left-hand side causes the pointer CURRENT to be dereferenced to produce its present target, namely
27 space to hold a cell (an integer and a cell pointer). The integer value on the right is copied to the integer
28 space on the left and the pointer component is pointer assigned (the descriptor on the right is copied
29 into the space for a descriptor on the left). When a statement such as

```
30 CURRENT => CURRENT % NEXT_CELL
```

31 is executed, the descriptor value in CURRENT % NEXT_CELL is copied to the descriptor named
32 CURRENT. In this case, CURRENT is made to point at a different target.

33 In the intrinsic assignment statement, the space associated with the current pointer does not change but
34 the values stored in that space do. In the pointer assignment, the current pointer is made to associate
35 with different space. Using the intrinsic assignment causes a linked list of cells to be moved up through
36 the current "window"; the pointer assignment causes the current pointer to be moved down through the
37 list of cells.

38 C.4.5 An example of a FORALL construct containing a WHERE construct

```
INTEGER :: A(5,5)
```

```

1
2 ...
3 FORALL (I = 1:5)
4     WHERE (A(I,:) == 0)
5         A(:,I) = I
6     ELSEWHERE (A(I,:) > 2)
7         A(I,:) = 6
8     END WHERE
9 END FORALL

```

10 If prior to execution of the FORALL, A has the value

```

11 A =      1  0  0  0  0
12          2  1  1  1  0
13          1  2  2  0  2
14          2  1  0  2  3
15          1  0  0  0  0

```

16 After execution of the assignment statements following the WHERE statement A has the value A'. The
 17 mask created from row one is used to mask the assignments to column one; the mask from row two is
 18 used to mask assignments to column two; etc.

```

19 A' =      1  0  0  0  0
20           1  1  1  1  5
21           1  2  2  4  5
22           1  1  3  2  5
23           1  2  0  0  5

```

24 The masks created for assignments following the ELSEWHERE statement are

```

25 .NOT. (A(I,:) == 0) .AND. (A'(I,:) > 2)

```

26 Thus the only elements affected by the assignments following the ELSEWHERE statement are A(3, 5)
 27 and A(4, 5). After execution of the FORALL construct, A has the value

```

28 A =      1  0  0  0  0
29           1  1  1  1  5
30           1  2  2  4  6
31           1  1  3  2  6
32           1  2  0  0  5

```

33 C.4.6 Examples of FORALL statements

34 Example 1:

```

35 FORALL (J=1:M, K=1:N) X(K, J) = Y(J, K)
   FORALL (K=1:N) X(K, 1:M) = Y(1:M, K)

```

1

2 These statements both copy columns 1 through N of array Y into rows 1 through N of array X. They
3 are equivalent to

4 `X(1:N, 1:M) = TRANSPOSE (Y(1:M, 1:N) )`

5 Example 2:

6 The following FORALL statement computes five partial sums of subarrays of J.

7 `J = (/ 1, 2, 3, 4, 5 /)`

8 `FORALL (K = 1:5) J(K) = SUM (J(1:K) )`

9 SUM is allowed in a FORALL because intrinsic functions are pure (12.7). After execution of the FORALL
10 statement, J is equal to [1, 3, 6, 10, 15].

11 Example 3:

12 `FORALL (I = 2:N-1) X(I) = (X(I-1) + 2*X(I) + X(I+1) ) / 4`

13 has the same effect as

14 `X(2:N-1) = (X(1:N-2) + 2*X(2:N-1) + X(3:N) ) / 4`

15 C.5 Clause 8 notes

16 C.5.1 Loop control

17 Fortran provides several forms of loop control:

- 18 (1) With an iteration count and a DO variable. This is the classic Fortran DO loop.
- 19 (2) Test a logical condition before each execution of the loop (DO WHILE).
- 20 (3) DO “forever”.

21 C.5.2 The CASE construct

22 At most one case block is selected for execution within a CASE construct, and there is no fall-through
23 from one block into another block within a CASE construct. Thus there is no requirement for the user
24 to exit explicitly from a block.

25 C.5.3 Examples of DO constructs

26 The following are all valid examples of block DO constructs.

27 Example 1:

```
28     SUM = 0.0
29     READ (IUN) N
30     OUTER: DO L = 1, N           ! A DO with a construct name
31         READ (IUN) IQUAL, M, ARRAY (1:M)
32         IF (IQUAL < IQUAL_MIN) CYCLE OUTER    ! Skip inner loop
33         INNER: DO 40 I = 1, M       ! A DO with a label and a name
```

```

1          CALL CALCULATE (ARRAY (I), RESULT)
2          IF (RESULT < 0.0) CYCLE
3          SUM = SUM + RESULT
4          IF (SUM > SUM_MAX) EXIT OUTER
5      40    END DO INNER
6          END DO OUTER

```

7 The outer loop has an iteration count of MAX (N, 0), and will execute that number of times or until
8 SUM exceeds SUM_MAX, in which case the EXIT OUTER statement terminates both loops. The inner
9 loop is skipped by the first CYCLE statement if the quality flag, IQUAL, is too low. If CALCULATE
10 returns a negative RESULT, the second CYCLE statement prevents it from being summed. Both loops
11 have construct names and the inner loop also has a label. A construct name is required in the EXIT
12 statement in order to terminate both loops, but is optional in the CYCLE statements because each
13 belongs to its innermost loop.

14 Example 2:

```

15      N = 0
16      DO 50, I = 1, 10
17          J = I
18          DO K = 1, 5
19              L = K
20              N = N + 1 ! This statement executes 50 times
21          END DO ! Nonlabeled DO inside a labeled DO
22      50 CONTINUE

```

23 After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

24 Example 3:

```

25      N = 0
26      DO I = 1, 10
27          J = I
28          DO 60, K = 5, 1 ! This inner loop is never executed
29              L = K
30              N = N + 1
31      60    CONTINUE ! Labeled DO inside a nonlabeled DO
32      END DO

```

33 After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by
34 these statements.

35 The following are all valid examples of nonblock DO constructs:

36 Example 4:

```

37      DO 70
38          READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X

```

```

1      IF (IOS /= 0) EXIT
2      IF (X < 0.) GOTO 70
3      CALL SUBA (X)
4      CALL SUBB (X)
5      ...
6      CALL SUBY (X)
7      CYCLE
8      70  CALL SUBNEG (X)  ! SUBNEG called only when X < 0.

```

9 This is not a block DO construct because it ends with a statement other than END DO or CONTINUE. The loop will
 10 continue to execute until an end-of-file condition or input/output error occurs.

11 Example 5:

```

12      SUM = 0.0
13      READ (IUN) N
14      DO 80, L = 1, N
15          READ (IUN) IQUAL, M, ARRAY (1:M)
16          IF (IQUAL < IQUAL_MIN) M = 0  ! Skip inner loop
17          DO 80 I = 1, M
18              CALL CALCULATE (ARRAY (I), RESULT)
19              IF (RESULT < 0.) CYCLE
20              SUM = SUM + RESULT
21              IF (SUM > SUM_MAX) GOTO 81
22      80  CONTINUE  ! This CONTINUE is shared by both loops
23      81 CONTINUE

```

24 This example is similar to Example 1 above, except that the two loops are not block DO constructs because they share
 25 the CONTINUE statement with the label 80. The terminal construct of the outer DO is the entire inner DO construct.
 26 The inner loop is skipped by forcing M to zero. If SUM grows too large, both loops are terminated by branching to the
 27 CONTINUE statement labeled 81. The CYCLE statement in the inner loop is used to skip negative values of RESULT.

28 Example 6:

```

29      N = 0
30      DO 100 I = 1, 10
31          J = I
32          DO 100 K = 1, 5
33              L = K
34      100  N = N + 1  ! This statement executes 50 times

```

35 In this example, the two loops share an assignment statement. After execution of this program fragment, I = 11, J = 10,
 36 K = 6, L = 5, and N = 50.

37 Example 7:

```

38      N = 0
39      DO 200 I = 1, 10
40          J = I
41          DO 200 K = 5, 1  ! This inner loop is never executed
42              L = K
43      200  N = N + 1

```

44 This example is very similar to the previous one, except that the inner loop is never executed. After execution of this
 45 program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

1 C.5.4 Examples of invalid DO constructs

2 The following are all examples of invalid skeleton DO constructs:

3 Example 1:

```
4 DO I = 1, 10
5     ...
6 END DO LOOP    ! No matching construct name
```

7 Example 2:

```
8 LOOP: DO 1000 I = 1, 10    ! No matching construct name
9     ...
10 1000 CONTINUE
```

11 Example 3:

```
12 LOOP1: DO
13     ...
14 END DO LOOP2    ! Construct names don't match
```

15 Example 4:

```
16 DO I = 1, 10    ! Label required or ...
17     ...
18 1010 CONTINUE    ! ... END DO required
```

19 Example 5:

```
20 DO 1020 I = 1, 10
21     ...
22 1021 END DO      ! Labels don't match
```

23 Example 6:

```
24 FIRST: DO I = 1, 10
25     SECOND: DO J = 1, 5
26         ...
27     END DO FIRST    ! Improperly nested DOs
28 END DO SECOND
```


1 C.6 Clause 9 notes

2 C.6.1 External files (9.2)

3 This part of ISO/IEC 1539 accommodates, but does not require, file cataloging. To do this, several
4 concepts are introduced.

5 C.6.1.1 File connection (9.4)

6 Before any input/output may be performed on a file, it shall be connected to a unit. The unit then serves
7 as a designator for that file as long as it is connected. To be connected does not imply that “buffers”
8 have or have not been allocated, that “file-control tables” have or have not been filled, or that any other
9 method of implementation has been used. Connection means that (barring some other fault) a READ
10 or WRITE statement may be executed on the unit, hence on the file. Without a connection, a READ
11 or WRITE statement shall not be executed.

12 C.6.1.2 File existence (9.2.1)

13 Totally independent of the connection state is the property of existence, this being a file property. The
14 processor “knows” of a set of files that exist at a given time for a given program. This set would include
15 tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set may exclude files inaccessible
16 to the program because of security, because they are already in use by another program, etc. This part
17 of ISO/IEC 1539 does not specify which files exist, hence wide latitude is available to a processor to
18 implement security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of
19 the files that a program can potentially process.

20 All four combinations of connection and existence may 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

21 Means are provided to create, delete, connect, and disconnect files.

22 C.6.1.3 File names (9.4.5.8)

23 A file may have a name. The form of a file name is not specified. If a system does not have some form of
24 cataloging or tape labeling for at least some of its files, all file names will disappear at the termination
25 of execution. This is a valid implementation. Nowhere does this part of ISO/IEC 1539 require names to
26 survive for any period of time longer than the execution time span of a program. Therefore, this part
27 of ISO/IEC 1539 does not impose cataloging as a prerequisite. The naming feature is intended to allow
28 use of a cataloging system where one exists.

29 C.6.1.4 File access (9.2.2)

30 This part of ISO/IEC 1539 does not address problems of security, protection, locking, and many other
31 concepts that may be part of the concept of “right of access”. Such concepts are considered to be in the
32 province of an operating system.

33 The OPEN and INQUIRE statements can be extended naturally to consider these things.

1 Possible access methods for a file are: sequential and direct. The processor may implement two different
2 types of files, each with its own access method. It might also implement one type of file with two different
3 access methods.

4 Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key
5 is a positive integer.

6 **C.6.2 Nonadvancing input/output (9.2.3.1)**

7 Data transfer statements affect the positioning of an external file. In FORTRAN 77, if no error or end-of-
8 file condition exists, the file is positioned after the record just read or written and that record becomes
9 the preceding record. This part of ISO/IEC 1539 contains the record positioning ADVANCE= specifier
10 in a data transfer statement that provides the capability of maintaining a position within the current
11 record from one formatted data transfer statement to the next data transfer statement. The value NO
12 provides this capability. The value YES positions the file after the record just read or written. The
13 default is YES.

14 The tab edit descriptor and the slash are still appropriate for use with this type of record access but the
15 tab will not reposition before the left tab limit.

16 A BACKSPACE of a file that is positioned within a record causes the specified unit to be positioned
17 before the current record.

18 If the last data transfer statement was WRITE and the file is positioned within a record, the file will be
19 positioned implicitly after the current record before an ENDFILE record is written to the file, that is, a
20 REWIND, BACKSPACE, or ENDFILE statement following a nonadvancing WRITE statement causes
21 the file to be positioned at the end of the current output record before the endfile record is written to
22 the file.

23 This part of ISO/IEC 1539 provides a SIZE= specifier to be used with nonadvancing data transfer
24 statements. The variable in the SIZE= specifier will contain the count of the number of characters that
25 make up the sequence of values read by the data edit descriptors in this input statement.

26 The count is especially helpful if there is only one list item in the input list because it will contain the
27 number of characters that were present for the item.

28 The EOR= specifier is provided to indicate when an end-of-record condition has been encountered
29 during a nonadvancing data transfer statement. The end-of-record condition is not an error condition.
30 If this specifier is present, the current input list item that required more characters than the record
31 contained will be padded with blanks if PAD= 'YES' is in effect. This means that the input list item
32 was successfully completed. The file will then be positioned after the current record. The IOSTAT=
33 specifier, if present, will be defined with the value of the named constant IOSTAT_EOR from the ISO_
34 FORTRAN_ENV module and the data transfer statement will be terminated. Program execution will
35 continue with the statement specified in the EOR= specifier. The EOR= specifier gives the capability
36 of taking control of execution when the end-of-record has been found. The *do-variables* in *io-implied-*
37 *dos* retain their last defined value and any remaining items in the *input-item-list* retain their definition
38 status when an end-of-record condition occurs. The SIZE= specifier, if present, will contain the number
39 of characters read with the data edit descriptors during this READ statement.

40 For nonadvancing input, the processor is not required to read partial records. The processor may read
41 the entire record into an internal buffer and make successive portions of the record available to successive
42 input statements.

43 In an implementation of nonadvancing input/output in which a nonadvancing write to a terminal device
44 causes immediate display of the output, such a write can be used as a mechanism to output a prompt.
45 In this case, the statement

1 WRITE (*, FMT='(A)', ADVANCE='NO') 'CONTINUE?(Y/N): '

2 would result in the prompt

3 CONTINUE?(Y/N):

4 being displayed with no subsequent line feed.

5 The response, which might be read by a statement of the form

6 READ (*, FMT='(A)') ANSWER

7 can then be entered on the same line as the prompt as in

8 CONTINUE?(Y/N): Y

9 This part of ISO/IEC 1539 does not require that an implementation of nonadvancing input/output
 10 operate in this manner. For example, an implementation of nonadvancing output in which the display
 11 of the output is deferred until the current record is complete is also standard-conforming. Such an
 12 implementation will not, however, allow a prompting mechanism of this kind to operate.

13 C.6.3 Asynchronous input/output

14 Rather than limit support for asynchronous input/output to what has been traditionally provided by
 15 facilities such as BUFFERIN/BUFFEROUT, this part of ISO/IEC 1539 builds upon existing Fortran
 16 syntax. This permits alternative approaches for implementing asynchronous input/output, and simplifies
 17 the task of adapting existing standard-conforming programs to use asynchronous input/output.

18 Not all processors will actually perform input/output asynchronously, nor will every processor that does
 19 be able to handle data transfer statements with complicated input/output item lists in an asynchronous
 20 manner. Such processors can still be standard-conforming. Hopefully, the documentation for each
 21 Fortran processor will describe when, if ever, input/output will be performed asynchronously.

22 This part of ISO/IEC 1539 allows for at least two different conceptual models for asynchronous in-
 23 put/output.

24 Model 1: the processor will perform asynchronous input/output when the item list is simple (perhaps
 25 one contiguous named array) and the input/output is unformatted. The implementation cost is reduced,
 26 and this is the scenario most likely to be beneficial on traditional “big-iron” machines.

27 Model 2: The processor is free to do any of the following:

- 28 (1) on output, create a buffer inside the input/output library, completely formatted, and then
 29 start an asynchronous write of the buffer, and immediately return to the next statement in
 30 the program. The processor is free to wait for previously issued WRITES, or not, or
- 31 (2) pass the input/output list addresses to another processor/process, which will process the
 32 list items independently of the processor that executes the user's code. The addresses of the
 33 list items must be computed before the asynchronous READ/WRITE statement completes.
 34 There is still an ordering requirement on list item processing to handle things like READ
 35 (...) N,(a(i),i=1,N).

36 This part of ISO/IEC 1539 allows a user to issue a large number of asynchronous input/output requests,
 37 without waiting for any of them to complete, and then wait for any or all of them. It may be impossible,
 38 and undesirable to keep track of each of these input/output requests individually.

39 It is not necessary for all requests to be tracked by the runtime library. If an ID= specifier does not
 40 appear in on a READ or WRITE statement, the runtime is free to forget about this particular request
 41 once it has successfully completed. If it gets an ERR or END condition, the processor is free to report

1 this during any input/output operation to that unit. When an ID= specifier is present, the processor's
2 runtime input/output library is required to keep track of any END or ERR conditions for that particular
3 input/output request. However, if the input/output request succeeds without any exceptional conditions
4 occurring, then the runtime can forget that ID= value if it wishes. Typically, a runtime might only keep
5 track of the last request made, or perhaps a very few. Then, when a user WAITs for a particular request,
6 either the library knows about it (and does the right thing with respect to error handling, etc.), or will
7 assume it is one of those requests that successfully completed and was forgotten about (and will just
8 return without signaling any end or error conditions). It is incumbent on the user to pass valid ID=
9 values. There is no requirement on the processor to detect invalid ID= values. There is of course,
10 a processor dependent limit on how many outstanding input/output requests that generate an end or
11 error condition can be handled before the processor runs out of memory to keep track of such conditions.
12 The restrictions on the SIZE= variables are designed to allow the processor to update such variables at
13 any time (after the request has been processed, but before the WAIT operation), and then forget about
14 them. That's why there is no SIZE= specifier allowed in the various WAIT operations. Only exceptional
15 conditions (errors or ends of files) are expected to be tracked by individual request by the runtime, and
16 then only if an ID= specifier was present. The END= and EOR= specifiers have not been added to all
17 statements that can be WAIT operations. Instead, the IOSTAT variable will have to be queried after a
18 WAIT operation to handle this situation. This choice was made because we expect the WAIT statement
19 to be the usual method of waiting for input/output to complete (and WAIT does support the END=
20 and EOR= specifiers). This particular choice is philosophical, and was not based on significant technical
21 difficulties.

22 Note that the requirement to set the IOSTAT variable correctly requires an implementation to remember
23 which input/output requests got an EOR condition, so that a subsequent wait operation will return the
24 correct IOSTAT value. This means there is a processor defined limit on the number of outstanding
25 nonadvancing input/output requests that got an EOR condition (constrained by available memory to
26 keep track of this information, similar to END/ERR conditions).

27 **C.6.4 OPEN statement (9.4.5)**

28 A file may become connected to a unit either by preconnection or by execution of an OPEN statement.
29 Preconnection is performed prior to the beginning of execution of a program by means external to For-
30 tran. For example, it may be done by job control action or by processor-established defaults. Execution
31 of an OPEN statement is not required to access preconnected files (9.4.4).

32 The OPEN statement provides a means to access existing files that are not preconnected. An OPEN
33 statement may be used in either of two ways: with a file name (open-by-name) and without a file name
34 (open-by-unit). A unit is given in either case. Open-by-name connects the specified file to the specified
35 unit. Open-by-unit connects a processor-dependent default file to the specified unit. (The default file
36 might or might not have a name.)

37 Therefore, there are three ways a file may become connected and hence processed: preconnection, open-
38 by-name, and open-by-unit. Once a file is connected, there is no means in standard Fortran to determine
39 how it became connected.

40 An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection
41 methods may be performed on a file that does not exist. When a unit is preconnected, writing the first
42 record creates the file. With the other two methods, execution of the OPEN statement creates the file.

43 When an OPEN statement is executed, the unit specified in the OPEN might or might not already be
44 connected to a file. If it is already connected to a file (either through preconnection or by a prior OPEN),
45 then omitting the FILE= specifier in the OPEN statement implies that the file is to remain connected
46 to the unit. Such an OPEN statement may be used to change the values of the blank interpretation
47 mode, decimal edit mode, pad mode, input/output rounding mode, delimiter mode, and sign mode.

1 If the value of the ACTION= specifier is WRITE, then READ statements shall not refer to this connection. ACTION = 'WRITE' does not restrict positioning by a BACKSPACE statement or positioning specified by the POSITION= specifier with the value APPEND. However, a BACKSPACE statement or an OPEN statement containing POSITION = 'APPEND' may fail if the processor requires reading of the file to achieve the positioning.

6 The following examples illustrate these rules. In the first example, unit 10 is preconnected to a SCRATCH file; the OPEN statement changes the value of PAD= to YES.

```

8 CHARACTER (LEN = 20) CH1
9 WRITE (10, '(A)') 'THIS IS RECORD 1'
10 OPEN (UNIT = 10, STATUS = 'OLD', PAD = 'YES')
11 REWIND 10
12 READ (10, '(A20)') CH1    ! CH1 now has the value
13                          ! 'THIS IS RECORD 1    '
```

14 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.

```

17 CHARACTER (LEN = 25) CH2, CH3
18 OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
19 CH2 = '''THIS STRING HAS QUOTES.'''
20      ! Quotes in string CH2
21 WRITE (12, *) CH2          ! Written with no delimiters
22 OPEN (12, DELIM = 'QUOTE') ! Now quote is the delimiter
23 REWIND 12
24 READ (12, *) CH3    ! CH3 now has the value
25                  ! 'THIS STRING HAS QUOTES.  '
```

26 The next example is invalid because it attempts to change the value of the STATUS= specifier.

```

27 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
28 WRITE (10, *) A, B, C
29 OPEN (10, STATUS = 'SCRATCH')    ! Attempts to make FRED
30                                ! a SCRATCH file
```

31 The previous example could be made valid by closing the unit first, as in the next example.

```

32 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
33 WRITE (10, *) A, B, C
34 CLOSE (10)
35 OPEN (10, STATUS = 'SCRATCH')    ! Opens a different
36                                ! SCRATCH file
```

C.6.5 Connection properties (9.4.3)

When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection, the following connection properties, among others, may be established.

- (1) An access method, which is sequential, direct, or stream, is established for the connection (9.4.5.1).
- (2) A form, which is formatted or unformatted, is established for a connection to a file that exists or is created by the connection. For a connection that results from execution of an OPEN statement, a default form (which depends on the access method, as described in 9.2.2) is established if no form is specified. For a preconnected file that exists, a form is established by preconnection. For a preconnected file that does not exist, a form may be established, or the establishment of a form may be delayed until the file is created (for example, by execution of a formatted or unformatted WRITE statement) (9.4.5.9).
- (3) A record length may be established. If the access method is direct, the connection establishes a record length that specifies the length of each record of the file. An existing file with records that are not all of equal length shall not be connected for direct access.
If the access method is sequential, records of varying lengths are permitted. In this case, the record length established specifies the maximum length of a record in the file (9.4.5.13).

A processor has wide latitude in adapting these concepts and actions to its own cataloging and job control conventions. Some processors may require job control action to specify the set of files that exist or that will be created by a program. Some processors may require no job control action prior to execution. This part of ISO/IEC 1539 enables processors to perform dynamic open, close, or file creation operations, but it does not require such capabilities of the processor.

The meaning of “open” in contexts other than Fortran may include such things as mounting a tape, console messages, spooling, label checking, security checking, etc. These actions may occur upon job control action external to Fortran, upon execution of an OPEN statement, or upon execution of the first read or write of the file. The OPEN statement describes properties of the connection to the file and might or might not cause physical activities to take place. It is a place for an implementation to define properties of a file beyond those required in standard Fortran.

C.6.6 CLOSE statement (9.4.6)

Similarly, the actions of dismounting a tape, protection, etc. of a “close” may be implicit at the end of a run. The CLOSE statement might or might not cause such actions to occur. This is another place to extend file properties beyond those of standard Fortran. Note, however, that the execution of a CLOSE statement on a unit followed by an OPEN statement on the same unit to the same file or to a different file is a permissible sequence of events. The processor shall not deny this sequence solely because the implementation chooses to do the physical act of closing the file at the termination of execution of the program.

C.7 Clause 10 notes

C.7.1 Number of records (10.4, 10.5, 10.8.2)

The number of records read by an explicitly formatted advancing input statement can be determined from the following rule: a record is read at the beginning of the format scan (even if the input list is empty), at each slash edit descriptor encountered in the format, and when a format rescan occurs at the end of the format.

The number of records written by an explicitly formatted advancing output statement can be determined from the following rule: a record is written when a slash edit descriptor is encountered in the format,

1 when a format rescan occurs at the end of the format, and at completion of execution of the output
2 statement (even if the output list is empty). Thus, the occurrence of n successive slashes between two
3 other edit descriptors causes $n - 1$ blank lines if the records are printed. The occurrence of n slashes at
4 the beginning or end of a complete format specification causes n blank lines if the records are printed.
5 However, a complete format specification containing n slashes ($n > 0$) and no other edit descriptors
6 causes $n + 1$ blank lines if the records are printed. For example, the statements

```
7 PRINT 3  
8 3 FORMAT (/)
```

9 will write two records that cause two blank lines if the records are printed.

10 C.7.2 List-directed input (10.10.3)

11 The following examples illustrate list-directed input. A blank character is represented by b.

12 Example 1:

13 Program:

```
14 J = 3  
15 READ *, I  
16 READ *, J
```

17 Sequential input file:

```
18 record 1: b1b,4bbbbbb  
19 record 2: ,2bbbbbbbbb
```

20 Result: $I = 1$, $J = 3$.

21 Explanation: The second READ statement reads the second record. The initial comma in the record
22 designates a null value; therefore, J is not redefined.

23 Example 2:

24 Program:

```
25 CHARACTER A *8, B *1  
26 READ *, A, B
```

27 Sequential input file:

```
28 record 1: 'bbbbbbbbb'  
29 record 2: 'QXY'b'Z'
```

30 Result: $A = 'bbbbbbbbb'$, $B = 'Q'$

31 Explanation: In the first record, the rightmost apostrophe is interpreted as delimiting the constant (it
32 cannot be the first of a pair of embedded apostrophes representing a single apostrophe because this

1 would involve the prohibited “splitting” of the pair by the end of a record); therefore, A is assigned
2 the character constant 'bbbbbbbb'. The end of a record acts as a blank, which in this case is a value
3 separator because it occurs between two constants.

4 **C.8 Clause 11 notes**

5 **C.8.1 Main program and block data program unit (11.1, 11.3)**

6 The name of the main program or of a block data program unit has no explicit use within the Fortran
7 language. It is available for documentation and for possible use by a processor.

8 A processor may implement an unnamed main program or unnamed block data program unit by assigning
9 it a default name. However, this name shall not conflict with any other global name in a standard-
10 conforming program. This might be done by making the default name one that is not permitted in a
11 standard-conforming program (for example, by including a character not normally allowed in names)
12 or by providing some external mechanism such that for any given program the default name can be
13 changed to one that is otherwise unused.

14 **C.8.2 Dependent compilation (11.2)**

15 This part of ISO/IEC 1539, like its predecessors, is intended to permit the implementation of conform-
16 ing processors in which a program can be broken into multiple units, each of which can be separately
17 translated in preparation for execution. Such processors are commonly described as supporting separate
18 compilation. There is an important difference between the way separate compilation can be imple-
19 mented under this part of ISO/IEC 1539 and the way it could be implemented under the FORTRAN
20 77 International Standard. Under the FORTRAN 77 standard, any information required to translate a
21 program unit was specified in that program unit. Each translation was thus totally independent of all
22 others. Under this part of ISO/IEC 1539, a program unit can use information that was specified in a
23 separate module and thus may be dependent on that module. The implementation of this dependency
24 in a processor may be that the translation of a program unit may depend on the results of translating
25 one or more modules. Processors implementing the dependency this way are commonly described as
26 supporting dependent compilation.

27 The dependencies involved here are new only in the sense that the Fortran processor is now aware of
28 them. The same information dependencies existed under the FORTRAN 77 International Standard, but
29 it was the programmer's responsibility to transport the information necessary to resolve them by making
30 redundant specifications of the information in multiple program units. The availability of separate but
31 dependent compilation offers several potential advantages over the redundant textual specification of
32 information.

- 33 (1) Specifying information at a single place in the program ensures that different program units
34 using that information will be translated consistently. Redundant specification leaves the
35 possibility that different information will erroneously be specified. Even if an INCLUDE line
36 is used to ensure that the text of the specifications is identical in all involved program units,
37 the presence of other specifications (for example, an IMPLICIT statement) may change the
38 interpretation of that text.
- 39 (2) During the revision of a program, it is possible for a processor to assist in determining
40 whether different program units have been translated using different (incompatible) versions
41 of a module, although there is no requirement that a processor provide such assistance.
42 Inconsistencies in redundant textual specification of information, on the other hand, tend
43 to be much more difficult to detect.
- 44 (3) Putting information in a module provides a way of packaging it. Without modules, redun-
45 dant specifications frequently shall be interleaved with other specifications in a program

1 unit, making convenient packaging of such information difficult.

2 (4) Because a processor may be implemented such that the specifications in a module are
3 translated once and then repeatedly referenced, there is the potential for greater efficiency
4 than when the processor shall translate redundant specifications of information in multiple
5 program units.

6 The exact meaning of the requirement that the public portions of a module be available at the time of
7 reference is processor dependent. For example, a processor could consider a module to be available only
8 after it has been compiled and require that if the module has been compiled separately, the result of
9 that compilation shall be identified to the compiler when compiling program units that use it.

10 C.8.2.1 USE statement and dependent compilation (11.2.2)

11 Another benefit of the USE statement is its enhanced facilities for name management. If one needs to
12 use only selected entities in a module, one can do so without having to worry about the names of all
13 the other entities in that module. If one needs to use two different modules that happen to contain
14 entities with the same name, there are several ways to deal with the conflict. If none of the entities with
15 the same name are to be used, they can simply be ignored. If the name happens to refer to the same
16 entity in both modules (for example, if both modules obtained it from a third module), then there is no
17 confusion about what the name denotes and the name can be freely used. If the entities are different
18 and one or both is to be used, the local renaming facility in the USE statement makes it possible to give
19 those entities different names in the program unit containing the USE statements.

20 A benefit of using the ONLY option consistently, as compared to USE without it, is that the module
21 from which each accessed entity is accessed is explicitly specified in each program unit. This means that
22 one need not search other program units to find where each one is defined. This reduces maintenance
23 costs.

24 A typical implementation of dependent but separate compilation may involve storing the result of trans-
25 lating a module in a file (or file element) whose name is derived from the name of the module. Note,
26 however, that the name of a module is limited only by the Fortran rules and not by the names allowed
27 in the file system. Thus the processor may have to provide a mapping between Fortran names and file
28 system names.

29 The result of translating a module could reasonably either contain only the information textually specified
30 in the module (with “pointers” to information originally textually specified in other modules) or contain
31 all information specified in the module (including copies of information originally specified in other
32 modules). Although the former approach would appear to save on storage space, the latter approach
33 can greatly simplify the logic necessary to process a USE statement and can avoid the necessity of
34 imposing a limit on the logical “nesting” of modules via the USE statement.

35 Variables declared in a module retain their definition status on much the same basis as variables in
36 a common block. That is, saved variables retain their definition status throughout the execution of a
37 program, while variables that are not saved retain their definition status only during the execution of
38 scoping units that reference the module. In some cases, it may be appropriate to put a USE statement
39 such as

40 `USE MY_MODULE, ONLY:`

41 in a scoping unit in order to assure that other procedures that it references can communicate through
42 the module. In such a case, the scoping unit would not access any entities from the module, but the
43 variables not saved in the module would retain their definition status throughout the execution of the
44 scoping unit.

45 There is an increased potential for undetected errors in a scoping unit that uses both implicit typing
46 and the USE statement. For example, in the program fragment

```

1  SUBROUTINE SUB
2      USE MY_MODULE
3      IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
4      X = F (B)
5      A = G (X) + H (X + 1)
6  END SUBROUTINE SUB

```

7 X could be either an implicitly typed real variable or a variable obtained from the module MY_MODULE
8 and might change from one to the other because of changes in MY_MODULE unrelated to the action
9 performed by SUB. Logic errors resulting from this kind of situation can be extremely difficult to locate.
10 Thus, the use of these features together is discouraged.

11 **C.8.2.2 Accessibility attributes**

12 The PUBLIC and PRIVATE attributes, which can be declared only in modules, divide the entities in a
13 module into those that are actually relevant to a scoping unit referencing the module and those that are
14 not. This information may be used to improve the performance of a Fortran processor. For example,
15 it may be possible to discard much of the information about the private entities once a module has
16 been translated, thus saving on both storage and the time to search it. Similarly, it may be possible
17 to recognize that two versions of a module differ only in the private entities they contain and avoid
18 retranslating program units that use that module when switching from one version of the module to the
19 other.

20 **C.8.3 Examples of the use of modules**

21 **C.8.3.1 Identical common blocks**

22 A common block and all its associated specification statements may be placed in a module named, for
23 example, MY_COMMON and accessed by a USE statement of the form

```
24 USE MY_COMMON
```

25 that accesses the whole module without any renaming. This ensures that all instances of the common
26 block are identical. Module MY_COMMON could contain more than one common block.

27 **C.8.3.2 Global data**

28 A module may contain only data objects, for example:

```

29 MODULE DATA_MODULE
30     SAVE
31     REAL A (10), B, C (20,20)
32     INTEGER :: I=0
33     INTEGER, PARAMETER :: J=10
34     COMPLEX D (J,J)
35 END MODULE DATA_MODULE

```

36 Data objects made global in this manner may have any combination of data types.

37 Access to some of these may be made by a USE statement with the ONLY option, such as:

1 USE DATA_MODULE, ONLY: A, B, D

2 and access to all of them may be made by the following USE statement:

3 USE DATA_MODULE

4 Access to all of them with some renaming to avoid name conflicts may be made by:

5 USE DATA_MODULE, AMODULE => A, DMODULE => D

6 C.8.3.3 Derived types

7 A derived type may be defined in a module and accessed in a number of program units. For example:

```
8 MODULE SPARSE
9     TYPE NONZERO
10        REAL A
11        INTEGER I, J
12    END TYPE NONZERO
13 END MODULE SPARSE
```

14 defines a type consisting of a real component and two integer components for holding the numerical
15 value of a nonzero matrix element and its row and column indices.

16 C.8.3.4 Global allocatable arrays

17 Many programs need large global allocatable arrays whose sizes are not known before program execution.

18 A simple form for such a program is:

```
19 PROGRAM GLOBAL_WORK
20     CALL CONFIGURE_ARRAYS      ! Perform the appropriate allocations
21     CALL COMPUTE               ! Use the arrays in computations
22 END PROGRAM GLOBAL_WORK
23 MODULE WORK_ARRAYS            ! An example set of work arrays
24     INTEGER N
25     REAL, ALLOCATABLE, SAVE :: A (:), B (:, :), C (:, :, :)
26 END MODULE WORK_ARRAYS
27 SUBROUTINE CONFIGURE_ARRAYS    ! Process to set up work arrays
28     USE WORK_ARRAYS
29     READ (*, *) N
30     ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
31 END SUBROUTINE CONFIGURE_ARRAYS
32 SUBROUTINE COMPUTE
33     USE WORK_ARRAYS
34     ... ! Computations involving arrays A, B, and C
35 END SUBROUTINE COMPUTE
```

36 Typically, many subprograms need access to the work arrays, and all such subprograms would contain
37 the statement

1 USE WORK_ARRAYS

2 C.8.3.5 Procedure libraries

3 Interface bodies for external procedures in a library may be gathered into a module. This permits the
4 use of argument keywords and optional arguments, and allows static checking of the references. Different
5 versions may be constructed for different applications, using argument keywords in common use in each
6 application.

7 An example is the following library module:

```
8 MODULE LIBRARY_LLS
9   INTERFACE
10     SUBROUTINE LLS (X, A, F, FLAG)
11       REAL X (:, :)
12       ! The SIZE in the next statement is an intrinsic function
13       REAL, DIMENSION (SIZE (X, 2)) :: A, F
14       INTEGER FLAG
15     END SUBROUTINE LLS
16     ...
17   END INTERFACE
18   ...
19 END MODULE LIBRARY_LLS
```

20 This module allows the subroutine LLS to be invoked:

```
21 USE LIBRARY_LLS
22   ...
23 CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
24   ...
```

25 C.8.3.6 Operator extensions

26 In order to extend an intrinsic operator symbol to have an additional meaning, an interface block
27 specifying that operator symbol in the OPERATOR option of the INTERFACE statement may be
28 placed in a module.

29 For example, // may be extended to perform concatenation of two derived-type objects serving as varying
30 length character strings and + may be extended to specify matrix addition for type MATRIX or interval
31 arithmetic addition for type INTERVAL.

32 A module might contain several such interface blocks. An operator may be defined by an external
33 function (either in Fortran or some other language) and its procedure interface placed in the module.

34 C.8.3.7 Data abstraction

35 In addition to providing a portable means of avoiding the redundant specification of information in
36 multiple program units, a module provides a convenient means of “packaging” related entities, such as
37 the definitions of the representation and operations of an abstract data type. The following example

1 of a module defines a data abstraction for a SET type where the elements of each set are of type
 2 integer. The usual set operations of UNION, INTERSECTION, and DIFFERENCE are provided.
 3 The CARDINALITY function returns the cardinality of (number of elements in) its set argument.
 4 Two functions returning logical values are included, ELEMENT and SUBSET. ELEMENT defines the
 5 operator .IN. and SUBSET extends the operator <=. ELEMENT determines if a given scalar integer
 6 value is an element of a given set, and SUBSET determines if a given set is a subset of another given
 7 set. (Two sets may be checked for equality by comparing cardinality and checking that one is a subset
 8 of the other, or checking to see if each is a subset of the other.)

9 The transfer function SETF converts a vector of integer values to the corresponding set, with duplicate
 10 values removed. Thus, a vector of constant values can be used as set constants. An inverse transfer
 11 function VECTOR returns the elements of a set as a vector of values in ascending order. In this SET
 12 implementation, set data objects have a maximum cardinality of 200.

```

13 MODULE INTEGER_SETS
14 ! This module is intended to illustrate use of the module facility
15 ! to define a new type, along with suitable operators.
16
17 INTEGER, PARAMETER :: MAX_SET_CARD = 200
18
19 TYPE SET                                ! Define SET type
20     PRIVATE
21     INTEGER CARD
22     INTEGER ELEMENT (MAX_SET_CARD)
23 END TYPE SET
24
25 INTERFACE OPERATOR (.IN.)
26     MODULE PROCEDURE ELEMENT
27 END INTERFACE OPERATOR (.IN.)
28
29 INTERFACE OPERATOR (<=)
30     MODULE PROCEDURE SUBSET
31 END INTERFACE OPERATOR (<=)
32
33 INTERFACE OPERATOR (+)
34     MODULE PROCEDURE UNION
35 END INTERFACE OPERATOR (+)
36
37 INTERFACE OPERATOR (-)
38     MODULE PROCEDURE DIFFERENCE
39 END INTERFACE OPERATOR (-)
40
41 INTERFACE OPERATOR (*)
42     MODULE PROCEDURE INTERSECTION
43 END INTERFACE OPERATOR (*)
44
45 CONTAINS

```

```

1
2  INTEGER FUNCTION CARDINALITY (A)      ! Returns cardinality of set A
3      TYPE (SET), INTENT (IN) :: A
4      CARDINALITY = A % CARD
5  END FUNCTION CARDINALITY
6
7  LOGICAL FUNCTION ELEMENT (X, A)        ! Determines if
8      INTEGER, INTENT(IN) :: X          ! element X is in set A
9      TYPE (SET), INTENT(IN) :: A
10     ELEMENT = ANY (A % ELEMENT (1 : A % CARD) == X)
11 END FUNCTION ELEMENT
12
13 FUNCTION UNION (A, B)                  ! Union of sets A and B
14     TYPE (SET) UNION
15     TYPE (SET), INTENT(IN) :: A, B
16     INTEGER J
17     UNION = A
18     DO J = 1, B % CARD
19         IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
20             IF (UNION % CARD < MAX_SET_CARD) THEN
21                 UNION % CARD = UNION % CARD + 1
22                 UNION % ELEMENT (UNION % CARD) = &
23                     B % ELEMENT (J)
24             ELSE
25                 ! Maximum set size exceeded . . .
26             END IF
27         END IF
28     END DO
29 END FUNCTION UNION
30
31 FUNCTION DIFFERENCE (A, B)             ! Difference of sets A and B
32     TYPE (SET) DIFFERENCE
33     TYPE (SET), INTENT(IN) :: A, B
34     INTEGER J, X
35     DIFFERENCE % CARD = 0               ! The empty set
36     DO J = 1, A % CARD
37         X = A % ELEMENT (J)
38         IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, X)
39     END DO
40 END FUNCTION DIFFERENCE
41
42 FUNCTION INTERSECTION (A, B)           ! Intersection of sets A and B
43     TYPE (SET) INTERSECTION
44     TYPE (SET), INTENT(IN) :: A, B
45     INTERSECTION = A - (A - B)

```

```

1  END FUNCTION INTERSECTION
2
3  LOGICAL FUNCTION SUBSET (A, B)           ! Determines if set A is
4      TYPE (SET), INTENT(IN) :: A, B      ! a subset of set B
5      INTEGER I
6      SUBSET = A % CARD <= B % CARD
7      IF (.NOT. SUBSET) RETURN             ! For efficiency
8      DO I = 1, A % CARD
9          SUBSET = SUBSET .AND. (A % ELEMENT (I) .IN. B)
10     END DO
11 END FUNCTION SUBSET
12
13 TYPE (SET) FUNCTION SETF (V)             ! Transfer function between a vector
14     INTEGER V (:)                        ! of elements and a set of elements
15     INTEGER J                            ! removing duplicate elements
16     SETF % CARD = 0
17     DO J = 1, SIZE (V)
18         IF (.NOT. (V (J) .IN. SETF)) THEN
19             IF (SETF % CARD < MAX_SET_CARD) THEN
20                 SETF % CARD = SETF % CARD + 1
21                 SETF % ELEMENT (SETF % CARD) = V (J)
22             ELSE
23                 ! Maximum set size exceeded . . .
24             END IF
25         END IF
26     END DO
27 END FUNCTION SETF
28
29 FUNCTION VECTOR (A)                      ! Transfer the values of set A
30     TYPE (SET), INTENT (IN) :: A         ! into a vector in ascending order
31     INTEGER, POINTER :: VECTOR (:)
32     INTEGER I, J, K
33     ALLOCATE (VECTOR (A % CARD))
34     VECTOR = A % ELEMENT (1 : A % CARD)
35     DO I = 1, A % CARD - 1               ! Use a better sort if
36         DO J = I + 1, A % CARD           ! A % CARD is large
37             IF (VECTOR (I) > VECTOR (J)) THEN
38                 K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
39             END IF
40         END DO
41     END DO
42 END FUNCTION VECTOR
43
44 END MODULE INTEGER_SETS

```

```

1  Examples of using INTEGER_SETS (A, B, and C are variables of type SET; X
2  is an integer variable):
3  ! Check to see if A has more than 10 elements
4  IF (CARDINALITY (A) > 10) ...
5
6  ! Check for X an element of A but not of B
7  IF (X .IN. (A - B)) ...
8
9  ! C is the union of A and the result of B intersected
10 ! with the integers 1 to 100
11 C = A + B * SETF ([I, I = 1, 100])
12
13 ! Does A have any even numbers in the range 1:100?
14 IF (CARDINALITY (A * SETF ([I, I = 2, 100, 2]))) > 0) ...
15
16 PRINT *, VECTOR (B) ! Print out the elements of set B, in ascending order

```

17 C.8.3.8 Public entities renamed

18 At times it may be necessary to rename entities that are accessed with USE statements. Care should be
 19 taken if the referenced modules also contain USE statements.

20 The following example illustrates renaming features of the USE statement.

```

21 MODULE J; REAL JX, JY, JZ; END MODULE J
22 MODULE K
23   USE J, ONLY : KX => JX, KY => JY
24   ! KX and KY are local names to module K
25   REAL KZ      ! KZ is local name to module K
26   REAL JZ      ! JZ is local name to module K
27 END MODULE K
28 PROGRAM RENAME
29   USE J; USE K
30   ! Module J's entity JX is accessible under names JX and KX
31   ! Module J's entity JY is accessible under names JY and KY
32   ! Module K's entity KZ is accessible under name KZ
33   ! Module J's entity JZ and K's entity JZ are different entities
34   ! and shall not be referenced
35   ...
36 END PROGRAM RENAME

```

37 C.8.4 Modules with submodules

38 Each submodule specifies that it is the child of exactly one parent module or submodule. Therefore, a
 39 module and all of its descendant submodules stand in a tree-like relationship one to another.

40 If a module procedure interface body that is specified in a module has public accessibility, and its

1 corresponding separate module procedure is defined in a descendant of that module, the procedure can
 2 be accessed by use association. No other entity in a submodule can be accessed by use association. Each
 3 program unit that references a module by use association depends on it, and each submodule depends
 4 on its ancestor module. Therefore, if one changes a separate module procedure body in a submodule but
 5 does not change its corresponding module procedure interface, a tool for automatic program translation
 6 would not need to reprocess program units that reference the module by use association. This is so
 7 even if the tool exploits the relative modification times of files as opposed to comparing the result of
 8 translating the module to the result of a previous translation.

9 By constructing taller trees, one can put entities at intermediate levels that are shared by submodules
 10 at lower levels; changing these entities cannot change the interpretation of anything that is accessible
 11 from the module by use association. Developers of modules that embody large complicated concepts
 12 can exploit this possibility to organize components of the concept into submodules, while preserving the
 13 privacy of entities that are shared by the submodules and that ought not to be exposed to users of the
 14 module. Putting these shared entities at an intermediate level also prevents cascades of reprocessing
 15 and testing if some of them are changed.

16 The following example illustrates a module, `color_points`, with a submodule, `color_points_a`, that in
 17 turn has a submodule, `color_points_b`. Public entities declared within `color_points` can be accessed
 18 by use association. The submodules `color_points_a` and `color_points_b` can be changed without
 19 causing retranslation of program units that reference the module `color_points`.

20 The module `color_points` does not have a *module-subprogram-part*, but a *module-subprogram-part* is
 21 not prohibited. The module could be published as definitive specification of the interface, without
 22 revealing trade secrets contained within `color_points_a` or `color_points_b`. Of course, a similar
 23 module without the `module` prefix in the interface bodies would serve equally well as documentation –
 24 but the procedures would be external procedures. It would make little difference to the consumer, but
 25 the developer would forfeit all of the advantages of modules.

```

26  module color_points
27
28      type color_point
29          private
30          real :: x, y
31          integer :: color
32      end type color_point
33
34      interface                ! Interfaces for procedures with separate
35                              ! bodies in the submodule color_points_a
36          module subroutine color_point_del ( p ) ! Destroy a color_point object
37              type(color_point), allocatable :: p
38          end subroutine color_point_del
39          ! Distance between two color_point objects
40          real module function color_point_dist ( a, b )
41              type(color_point), intent(in) :: a, b
42          end function color_point_dist
43          module subroutine color_point_draw ( p ) ! Draw a color_point object
44              type(color_point), intent(in) :: p
45          end subroutine color_point_draw
46          module subroutine color_point_new ( p ) ! Create a color_point object
```

```

1      type(color_point), allocatable :: p
2      end subroutine color_point_new
3  end interface
4
5  end module color_points

```

6 The only entities within `color_points_a` that can be accessed by use association are separate module
7 procedures for which corresponding module procedure interface bodies are provided in `color_points`.
8 If the procedures are changed but their interfaces are not, the interface from program units that access
9 them by use association is unchanged. If the module and submodule are in separate files, utilities that
10 examine the time of modification of a file would notice that changes in the module could affect the
11 translation of its submodules or of program units that reference the module by use association, but
12 that changes in submodules could not affect the translation of the parent module or program units that
13 reference it by use association.

14 The variable `instance_count` is not accessible by use association of `color_points`, but is accessible
15 within `color_points_a`, and its submodules.

```

16  submodule ( color_points ) color_points_a ! Submodule of color_points
17
18      integer, save :: instance_count = 0
19
20      interface                                ! Interface for a procedure with a separate
21                                          ! body in submodule color_points_b
22      module subroutine inquire_palette ( pt, pal )
23          use palette_stuff                ! palette_stuff, especially submodules
24                                          ! thereof, can reference color_points by use
25                                          ! association without causing a circular
26                                          ! dependence during translation because this
27                                          ! use is not in the module. Furthermore,
28                                          ! changes in the module palette_stuff do not
29                                          ! affect the translation of color_points.
30          type(color_point), intent(in) :: pt
31          type(palette), intent(out) :: pal
32      end subroutine inquire_palette
33
34  end interface
35
36  contains ! Invisible bodies for public module procedure interfaces
37      ! declared in the module
38
39      module subroutine color_point_del ( p )
40          type(color_point), allocatable :: p
41          instance_count = instance_count - 1
42          deallocate ( p )
43      end subroutine color_point_del
44
45      real module function color_point_dist ( a, b ) result ( dist )

```

```

1
2     type(color_point), intent(in) :: a, b
3     dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
4 end function color_point_dist
5 module subroutine color_point_new ( p )
6     type(color_point), allocatable :: p
7     instance_count = instance_count + 1
8     allocate ( p )
9 end subroutine color_point_new
10
11 end submodule color_points_a

```

12 The subroutine `inquire_palette` is accessible within `color_points_a` because its interface is declared
13 therein. It is not, however, accessible by use association, because its interface is not declared in the
14 module, `color_points`. Since the interface is not declared in the module, changes in the interface
15 cannot affect the translation of program units that reference the module by use association.

```

16 submodule ( color_points:color_points_a ) color_points_b ! Subsidiary**2 submodule
17
18 contains
19     ! Invisible body for interface declared in the ancestor module
20     module subroutine color_point_draw ( p )
21         use palette_stuff, only: palette
22         type(color_point), intent(in) :: p
23         type(palette) :: MyPalette
24         ...; call inquire_palette ( p, MyPalette ); ...
25     end subroutine color_point_draw
26
27     ! Invisible body for interface declared in the parent submodule
28     module procedure inquire_palette
29         ... implementation of inquire_palette
30     end procedure inquire_palette
31
32     subroutine private_stuff ! not accessible from color_points_a
33         ...
34     end subroutine private_stuff
35
36 end submodule color_points_b
37
38 module palette_stuff
39     type :: palette ; ... ; end type palette
40 contains
41     subroutine test_palette ( p )
42         ! Draw a color wheel using procedures from the color_points module
43         type(palette), intent(in) :: p
44         use color_points ! This does not cause a circular dependency because

```

```

1
2           ! the "use palette_stuff" that is logically within
3           ! color_points is in the color_points_a submodule.
4       ...
5       end subroutine test_palette
6   end module palette_stuff

```

7 There is a use palette_stuff in color_points_a, and a use color_points in palette_stuff. The
8 use palette_stuff would cause a circular reference if it appeared in color_points. In this case, it
9 does not cause a circular dependence because it is in a submodule. Submodules cannot be referenced
10 by use association, and therefore what would be a circular appearance of use palette_stuff is not
11 accessed.

```

12   program main
13       use color_points
14       ! "instance_count" and "inquire_palette" are not accessible here
15       ! because they are not declared in the "color_points" module.
16       ! "color_points_a" and "color_points_b" cannot be referenced by
17       ! use association.
18       interface draw                                ! just to demonstrate it's possible
19           module procedure color_point_draw
20       end interface
21       type(color_point) :: C_1, C_2
22       real :: RC
23       ...
24       call color_point_new (c_1)                    ! body in color_points_a, interface in color_points
25       ...
26       call draw (c_1)                               ! body in color_points_b, specific interface
27                                           ! in color_points, generic interface here.
28       ...
29       rc = color_point_dist (c_1, c_2) ! body in color_points_a, interface in color_points
30       ...
31       call color_point_del (c_1)                    ! body in color_points_a, interface in color_points
32       ...
33   end program main

```

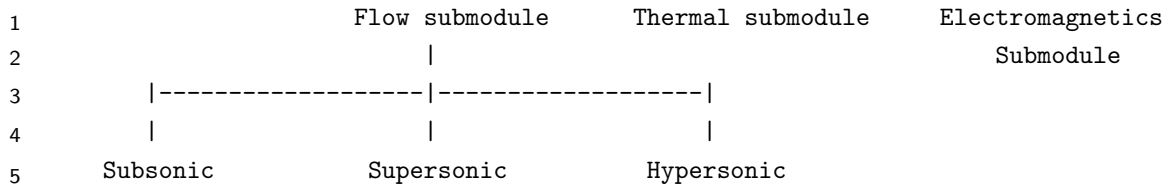
34 A multilevel submodule system can be used to package and organize a large and interconnected concept
35 without exposing entities of one subsystem to other subsystems.

36 Consider a Plasma module from a Tokamak simulator. A plasma simulation requires attention at least to
37 fluid flow, thermodynamics, and electromagnetism. Fluid flow simulation requires simulation of subsonic,
38 supersonic, and hypersonic flow. This problem decomposition can be reflected in the submodule structure
39 of the Plasma module:

```

40                               Plasma module
41                               |
42   |-----|-----|-----|
43   |               |               |

```



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** modules, 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 these entities are not accessible by use association, if any of them are changed, the new values cannot be accessed in program units that reference the **Plasma** module by use association; the answer to the question “where are these entities used” is therefore confined to the set of descendant submodules of the **Flow** submodule.

C.9 Clause 12 notes

C.9.1 Portability problems with external procedures (12.4.3.5)

There is a potential portability problem in a scoping unit that references an external procedure without explicitly declaring it to have the **EXTERNAL** attribute (5.3.8). On a different processor, the name of that procedure may be the name of a nonstandard intrinsic procedure and the processor would be permitted to interpret those procedure references as references to that intrinsic procedure. (On that processor, the program would also be viewed as not conforming to this part of ISO/IEC 1539 because of the references to the nonstandard intrinsic procedure.) Declaration of the **EXTERNAL** attribute causes the references to be to the external procedure regardless of the availability of an intrinsic procedure with the same name. Note that declaration of the type of a procedure is not enough to make it external, even if the type is inconsistent with the type of the result of an intrinsic of the same name.

C.9.2 Procedures defined by means other than Fortran (12.6.3)

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).

Procedures defined by means other than Fortran are considered external procedures because their definitions are not in a Fortran program unit and because they are referenced using global names. The use of the term external should not be construed as any kind of restriction on the way in which these procedures may be defined. For example, if the means other than Fortran has its own facilities for internal and external procedures, it is permissible to use them. If the means other than Fortran can create an “internal” procedure with a global name, it is permissible for such an “internal” procedure to be considered by Fortran to be an external procedure. The means other than Fortran for defining external procedures, including any restrictions on the structure for organization of those procedures, are entirely processor dependent.

A Fortran processor may limit its support of procedures defined by means other than Fortran such that these procedures may affect entities in the Fortran environment only on the same basis as procedures written in Fortran. For example, it might prohibit the value of a local variable from being changed by a procedure reference unless that variable were one of the arguments to the procedure.

C.9.3 Procedure interfaces (12.4)

1 In FORTRAN 77, the interface to an external procedure was always deduced from the form of references
 2 to that procedure and any declarations of the procedure name in the referencing program unit. In this
 3 part of ISO/IEC 1539, features such as argument keywords and optional arguments make it impossible
 4 to deduce sufficient information about the dummy arguments from the nature of the actual arguments
 5 to be associated with them, and features such as array function results and pointer function results
 6 make necessary extensions to the declaration of a procedure that cannot be done in a way that would
 7 be analogous with the handling of such declarations in FORTRAN 77. Hence, mechanisms are provided
 8 through which all the information about a procedure's interface may be made available in a scoping
 9 unit that references it. A procedure whose interface shall be deduced as in FORTRAN 77 is described
 10 as having an implicit interface. A procedure whose interface is fully known is described as having an
 11 explicit interface.

12 A scoping unit is allowed to contain an interface body for a procedure that does not exist in the program,
 13 provided the procedure described is never referenced or used in any other way. The purpose of this rule is
 14 to allow implementations in which the use of a module providing interface bodies describing the interface
 15 of every routine in a library would not automatically cause each of those library routines to be a part of
 16 the program referencing the module. Instead, only those library procedures actually referenced would
 17 be a part of the program. (In implementation terms, the mere presence of an interface body would not
 18 generate an external reference in such an implementation.)

19 **C.9.4 Abstract interfaces (12.4) and procedure pointer components (4.5)**

20 This is an example of a library module providing lists of callbacks that the user may register and invoke.

```

21 MODULE callback_list_module
22   !
23   ! Type for users to extend with their own data, if they so desire
24   !
25   TYPE callback_data
26   END TYPE
27   !
28   ! Abstract interface for the callback procedures
29   !
30   ABSTRACT INTERFACE
31     SUBROUTINE callback_procedure(data)
32       IMPORT callback_data
33       CLASS(callback_data),OPTIONAL :: data
34     END SUBROUTINE
35   END INTERFACE
36   !
37   ! The callback list type.
38   !
39   TYPE callback_list
40     PRIVATE
41     CLASS(callback_record),POINTER :: first => NULL()
42   END TYPE
43   !
44   ! Internal: each callback registration creates one of these
45   !

```

```

1  TYPE,PRIVATE :: callback_record
2      PROCEDURE(callback_procedure),POINTER,NOPASS :: proc
3      CLASS(callback_record),POINTER :: next
4      CLASS(callback_data),POINTER :: data => NULL();
5  END TYPE
6  PRIVATE invoke,forward_invoke
7  CONTAINS
8      !
9      ! Register a callback procedure with optional data
10     !
11     SUBROUTINE register_callback(list, entry, data)
12         TYPE(callback_list),INTENT(INOUT) :: list
13         PROCEDURE(callback_procedure) :: entry
14         CLASS(callback_data),OPTIONAL :: data
15         TYPE(callback_record),POINTER :: new,last
16         ALLOCATE(new)
17         new%proc => entry
18         IF (PRESENT(data)) ALLOCATE(new%data,SOURCE=data)
19         new%next => list%first
20         list%first => new
21     END SUBROUTINE
22     !
23     ! Internal: Invoke a single callback and destroy its record
24     !
25     SUBROUTINE invoke(callback)
26         TYPE(callback_record),POINTER :: callback
27         IF (ASSOCIATED(callback%data) THEN
28             CALL callback%proc(list%first%data)
29             DEALLOCATE(callback%data)
30         ELSE
31             CALL callback%proc
32         END IF
33         DEALLOCATE(callback)
34     END SUBROUTINE
35     !
36     ! Call the procedures in reverse order of registration
37     !
38     SUBROUTINE invoke_callback_reverse(list)
39         TYPE(callback_list),INTENT(INOUT) :: list
40         TYPE(callback_record),POINTER :: next,current
41         current => list%first
42         NULLIFY(list%first)
43         DO WHILE (ASSOCIATED(current))
44             next => current%next
45             CALL invoke(current)

```

```

1      current => next
2      END DO
3  END SUBROUTINE
4      !
5      ! Internal: Forward mode invocation
6      !
7      RECURSIVE SUBROUTINE forward_invoke(callback)
8          IF (ASSOCIATED(callback%next)) CALL forward_invoke(callback%next)
9          CALL invoke(callback)
10     END SUBROUTINE
11     !
12     ! Call the procedures in forward order of registration
13     !
14     SUBROUTINE invoke_callback_forward(list)
15         TYPE(callback_list),INTENT(INOUT) :: list
16         IF (ASSOCIATED(list%first)) CALL forward_invoke(list%first)
17     END SUBROUTINE
18 END

```

19 C.9.5 Argument association and evaluation (12.5.2)

20 There is a significant difference between the argument association allowed in this part of ISO/IEC 1539
21 and that supported by FORTRAN 77 and FORTRAN 66. In FORTRAN 77 and 66, actual arguments
22 were limited to consecutive storage units. With the exception of assumed length character dummy
23 arguments, the structure imposed on that sequence of storage units was always determined in the invoked
24 procedure and not taken from the actual argument. Thus it was possible to implement FORTRAN 66
25 and FORTRAN 77 argument association by supplying only the location of the first storage unit (except
26 for character arguments, where the length would also have to be supplied). However, this part of
27 ISO/IEC 1539 allows arguments that do not reside in consecutive storage locations (for example, an
28 array section), and dummy arguments that assume additional structural information from the actual
29 argument (for example, assumed-shape dummy arguments). Thus, the mechanism to implement the
30 argument association allowed in this part of ISO/IEC 1539 needs to be more general.

31 Because there are practical advantages to a processor that can support references to and from procedures
32 defined by a FORTRAN 77 processor, requirements for explicit interfaces make it possible to determine
33 whether a simple (FORTRAN 66/FORTRAN 77) argument association implementation mechanism is suffi-
34 cient or whether the more general mechanism is necessary (12.4.2). Thus a processor can be implemented
35 whose procedures expect the simple mechanism to be used whenever the procedure's interface is one that
36 uses only FORTRAN 77 features and that expects the more general mechanism otherwise (for example, if
37 there are assumed-shape or optional arguments). At the point of reference, the appropriate mechanism
38 can be determined from the interface if it is explicit and can be assumed to be the simple mechanism
39 if it is not. Note that if the simple mechanism is determined to be what the procedure expects, it may
40 be necessary for the processor to allocate consecutive temporary storage for the actual argument, copy
41 the actual argument to the temporary storage, reference the procedure using the temporary storage in
42 place of the actual argument, copy the contents of temporary storage back to the actual argument, and
43 deallocate the temporary storage.

44 While this is the particular implementation method these rules were designed to support, it is not
45 the only one possible. For example, on some processors, it may be possible to implement the general
46 argument association in such a way that the information involved in FORTRAN 77 argument association
47 may be found in the same places and the "extra" information is placed so it does not disturb a procedure

1 expecting only FORTRAN 77 argument association. With such an implementation, argument association
2 could be translated without regard to whether the interface is explicit or implicit.

3 The provisions for expression evaluation give the processor considerable flexibility for obtaining expres-
4 sion values in the most efficient way possible. This includes not evaluating or only partially evaluating an
5 operand, for example, if the value of the expression can be determined otherwise (7.1.7). This flexibility
6 applies to function argument evaluation, including the order of argument evaluation, delaying argument
7 evaluation, and omitting argument evaluation. A processor may delay the evaluation of an argument
8 in a procedure reference until the execution of the procedure refers to the value of that argument, pro-
9 vided delaying the evaluation of the argument does not otherwise affect the results of the program. The
10 processor may, with similar restrictions, entirely omit the evaluation of an argument not referenced in
11 the execution of the procedure. This gives processors latitude for optimization (for example, for parallel
12 processing).

13 C.9.6 Pointers and targets as arguments (12.5.2.5, 12.5.2.7, 12.5.2.8)

J3 internal note

Unresolved Technical Issue 082

The following paragraph takes no account of auto-targetting. Rewrite or delete.

14 If a dummy argument is declared to be a pointer, it may be matched only by an actual argument that
15 also is a pointer, and the characteristics of both arguments shall agree. A model for such an association is
16 that descriptor values of the actual pointer are copied to the dummy pointer. If the actual pointer has an
17 associated target, this target becomes accessible via the dummy pointer. If the dummy pointer becomes
18 associated with a different target during execution of the procedure, this target will be accessible via the
19 actual pointer after the procedure completes execution. If the dummy pointer becomes associated with
20 a local target that ceases to exist when the procedure completes, the actual pointer will be left dangling
21 in an undefined state. Such dangling pointers shall not be used.

22 When execution of a procedure completes, any pointer that remains defined and that is associated with
23 a dummy argument that has the TARGET attribute and is either a scalar or an assumed-shape array,
24 remains associated with the corresponding actual argument if the actual argument has the TARGET
25 attribute and is not an array section with a vector subscript.

```
26 REAL, POINTER      :: PBEST
27 REAL, TARGET       :: B (10000)
28 CALL BEST (PBEST, B)      ! Upon return PBEST is associated
29 ...                      ! with the 'best' element of B
30 CONTAINS
31   SUBROUTINE BEST (P, A)
32     REAL, POINTER, INTENT (OUT) :: P
33     REAL, TARGET, INTENT (IN)  :: A (:)
34     ...                        ! Find the 'best' element A(I)
35     P => A (I)
36   RETURN
37 END SUBROUTINE BEST
38 END
```

39 When procedure BEST completes, the pointer PBEST is associated with an element of B.

40 An actual argument without the TARGET attribute can become associated with a dummy argument

1 with the TARGET attribute. This permits pointers to become associated with the dummy argument
 2 during execution of the procedure that contains the dummy argument. For example:

```

3  INTEGER LARGE(100,100)
4  CALL SUB (LARGE)
5      ...
6  CALL SUB ( )
7  CONTAINS
8      SUBROUTINE SUB(ARG)
9          INTEGER, TARGET, OPTIONAL :: ARG(100,100)
10         INTEGER, POINTER, DIMENSION(:,:) :: PARG
11         IF (PRESENT(ARG)) THEN
12             PARG => ARG
13         ELSE
14             ALLOCATE (PARG(100,100))
15             PARG = 0
16         ENDIF
17         ... ! Code with lots of references to PARG
18         IF (.NOT. PRESENT(ARG)) DEALLOCATE(PARG)
19     END SUBROUTINE SUB
20 END

```

21 Within subroutine SUB the pointer PARG is either associated with the dummy argument ARG or it is
 22 associated with an allocated target. The bulk of the code can reference PARG without further calls to
 23 the PRESENT intrinsic.

24 C.9.7 Polymorphic Argument Association (12.5.2.10)

25 The following example illustrates polymorphic argument association rules using the derived types defined
 26 in Note 4.58.

```

27  TYPE(POINT) :: T2
28  TYPE(COLOR_POINT) :: T3
29  CLASS(POINT) :: P2
30  CLASS(COLOR_POINT) :: P3
31  ! Dummy argument is polymorphic and actual argument is of fixed type
32  SUBROUTINE SUB2 ( X2 ); CLASS(POINT) :: X2; ...
33  SUBROUTINE SUB3 ( X3 ); CLASS(COLOR_POINT) :: X3; ...
34
35  CALL SUB2 ( T2 ) ! Valid -- The declared type of T2 is the same as the
36                   !           declared type of X2.
37  CALL SUB2 ( T3 ) ! Valid -- The declared type of T3 is extended from
38                   !           the declared type of X2.
39  CALL SUB3 ( T2 ) ! Invalid -- The declared type of T2 is neither the
40                   !           same as nor extended from the declared type
41                   !           type of X3.

```

```

1
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.

```

41 C.10 Clause 13 notes

1 C.10.1 Module for THIS_IMAGE and IMAGE_INDEX

2 The intrinsics THIS_IMAGE (CO_ARRAY) and IMAGE_INDEX (CO_ARRAY, SUB) cannot be written
3 in Fortran since CO_ARRAY may be of any type and THIS_IMAGE (CO_ARRAY) needs to know the
4 index of the image on which the code is running.

5 As an example, here are simple versions that require the co-bounds to be specified as integer arrays and
6 require the image index for THIS_IMAGE (CO_ARRAY).

```

7 MODULE index
8 CONTAINS
9     INTEGER FUNCTION image_index(lbound, ubound, sub)
10         INTEGER, INTENT(IN) :: lbound(:), ubound(:), sub(:)
11         INTEGER              :: i, n
12         n = SIZE(sub)
13         image_index = sub(n) - lbound(n)
14         DO i = n-1, 1, -1
15             image_index = image_index*(ubound(i)-lbound(i)+1) + sub(i) - lbound(i)
16         END DO
17         image_index = image_index + 1
18     END FUNCTION image_index
19
20     INTEGER FUNCTION this_image(lbound, ubound, me) RESULT(sub)
21         INTEGER, INTENT(IN) :: lbound(:), ubound(:), me
22         INTEGER              :: sub(SIZE(lbound))
23         INTEGER              :: extent, i, m, ml, n
24         n = SIZE(sub)
25         m = me - 1
26         DO i = 1, n-1
27             extent = ubound(i) - lbound(i) + 1
28             ml = m
29             m = m/extent
30             sub(i) = ml - m*extent + lbound(i)
31         END DO
32         sub(n) = m + lbound(n)
33     END FUNCTION this_IMAGE
34 END MODULE index

```

35 C.10.2 Collective co-array subroutine variations

36 The subroutines of 13.5.15 return an array of the same shape as the given co-array after having applied
37 an operation on the images involved. Simple routines can be written to also apply the operation to the
38 elements of the co-array on an image. Various versions of a global sum can be programmed, for example:

```

39 MODULE global_sum_module
40     INTRINSIC, PRIVATE :: CO_SUM, SIZE, SUM
41 CONTAINS

```

```

1  REAL FUNCTION global_sum(array)
2      REAL,INTENT(IN) :: array(:, :)[*]
3      REAL,SAVE      :: temp[*]
4      temp = SUM(array)          ! Sum on the executing image
5      CALL CO_SUM(temp, global_sum)
6  END FUNCTION global_sum
7
8  REAL FUNCTION global_sum_mask(array, mask)
9      REAL,INTENT(IN)  :: array(:, :)[*]
10     LOGICAL,INTENT(IN) :: mask(:, :)
11     REAL,SAVE        :: temp[*]
12     temp = SUM(array, MASK=mask) ! Sum on the executing image
13     CALL CO_SUM(temp, global_sum_mask)
14 END FUNCTION global_sum_mask
15
16 FUNCTION global_sum_dim(array, dim)
17     REAL, INTENT(IN)  :: array(:, :)[*]
18     INTEGER, INTENT(IN) :: dim
19     REAL, ALLOCATABLE :: global_sum_dim(:)
20     REAL, ALLOCATABLE :: temp(:)[:]
21     ALLOCATE (global_sum_dim(SIZE(array, 3-dim)))
22     ALLOCATE (temp(SIZE(array, 3-dim))[*])
23     temp = SUM(array, dim)      ! Sum of the local part of the co-array.
24     CALL CO_SUM(temp, global_sum_dim)
25 END FUNCTION global_sum_dim
26 END MODULE global_sum_module

```

27 C.11 Clause 15 notes

28 C.11.1 Runtime environments

29 This part of ISO/IEC 1539 allows programs to contain procedures defined by means other than Fortran.
30 That raises the issues of initialization of and interaction between the runtime environments involved.

31 Implementations are free to solve these issues as they see fit, provided that

- 32 (1) heap allocation/deallocation (e.g., (DE)ALLOCATE in a Fortran subprogram and mal-
33 loc/free in a C function) can be performed without interference,
- 34 (2) input/output to and from external files can be performed without interference, as long as
35 procedures defined by different means do not do input/output with the same external file,
- 36 (3) input/output preconnections exist as required by the respective standards, and
- 37 (4) initialized data is initialized according to the respective standards.

38 C.11.2 Examples of Interoperation between Fortran and C Functions

39 The following examples illustrate the interoperation of Fortran and C functions. Two examples are
40 shown: one of Fortran calling C, and one of C calling Fortran. In each of the examples, the correspon-
41 dences of Fortran actual arguments, Fortran dummy arguments, and C formal parameters are described.

1 C.11.2.1 Example of Fortran calling C

2 C Function Prototype:

```
3      int C_Library_Function(void* sendbuf, int sendcount,
4                          int *recvcounts);
```

5 Fortran Modules:

```
6      MODULE FTN_C_1
7          USE, INTRINSIC :: ISO_C_BINDING
8      END MODULE FTN_C_1
9
10     MODULE FTN_C_2
11         INTERFACE
12
13             INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
14             (SENDBUF, SENDCOUNT, RECVCOUNTS)    &
15             BIND(C, NAME='C_Library_Function')
16             USE FTN_C_1
17             IMPLICIT NONE
18             TYPE (C_PTR), VALUE :: SENDBUF
19             INTEGER (C_INT), VALUE :: SENDCOUNT
20             TYPE (C_PTR), VALUE :: RECVCOUNTS
21         END FUNCTION C_LIBRARY_FUNCTION
22     END INTERFACE
23 END MODULE FTN_C_2
```

23 The module FTN_C.2 contains the declaration of the Fortran dummy arguments, which correspond to
 24 the C formal parameters. The intrinsic module ISO_C_BINDING is referenced in the module FTN_C.1.
 25 The NAME specifier is used in the BIND attribute in order to handle the case-sensitive name change
 26 between Fortran and C from 'C_LIBRARY_FUNCTION' to 'C_Library_Function'. See also Note 12.47.

27 The first C formal parameter is the pointer to void `sendbuf`, which corresponds to the Fortran dummy
 28 argument `SENDBUF`, which has the type `C_PTR` and the `VALUE` attribute.

29 The second C formal parameter is the int `sendcount`, which corresponds to the Fortran dummy argument
 30 `SENDCOUNT`, which has the type `INTEGER(C_INT)` and the `VALUE` attribute.

31 The third C formal parameter is the pointer to int `recvcounts`, which corresponds to the Fortran dummy
 32 argument `RECVCOUNTS`, which has the type `C_PTR` and the `VALUE` attribute.

33 Fortran Calling Sequence:

```
34      USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
35      USE FTN_C_2
36      ...
      REAL (C_FLOAT), TARGET :: SEND(100)
```

```

1
2      INTEGER (C_INT)          :: SENDCOUNT
3      INTEGER (C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(100)
4      ...
5      ALLOCATE( RECVCOUNTS(100) )
6      ...
7      CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
8      C_LOC(RECVCOUNTS))
9      ...

```

10 The preceding code contains the declaration of the Fortran actual arguments associated with the above-
 11 listed Fortran dummy arguments.

12 The first Fortran actual argument is the address of the first element of the array SEND, which has the
 13 type REAL(C_FLOAT) and the TARGET attribute. This address is returned by the intrinsic function
 14 C_LOC. This actual argument is associated with the Fortran dummy argument SENDBUF, which has
 15 the type C_PTR and the VALUE attribute.

16 The second Fortran actual argument is SENDCOUNT of type INTEGER(C_INT), which is associated
 17 with the Fortran dummy argument SENDCOUNT, which has the type INTEGER(C_INT) and the
 18 VALUE attribute.

19 The third Fortran actual argument is the address of the first element of the allocatable array RECV-
 20 COUNTS, with has the type REAL(C_FLOAT) and the TARGET attribute. This address is returned
 21 by the intrinsic function C_LOC. This actual argument is associated with the Fortran dummy argument
 22 RECVCOUNTS, which has the type C_PTR and the VALUE attribute.

23 C.11.2.2 Example of C calling Fortran

24 Fortran Code:

```

25 SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS) BIND(C)
26   USE, INTRINSIC :: ISO_C_BINDING
27   IMPLICIT NONE
28   INTEGER (C_LONG), VALUE          :: ALPHA
29   REAL (C_DOUBLE), INTENT(INOUT)   :: BETA
30   INTEGER (C_LONG), INTENT(OUT)     :: GAMMA
31   REAL (C_DOUBLE), DIMENSION(*), INTENT(IN) :: DELTA
32   TYPE, BIND(C) :: PASS
33     INTEGER (C_INT) :: LENC, LENF
34     TYPE (C_PTR)    :: C, F
35   END TYPE PASS
36   TYPE (PASS), INTENT(INOUT) :: ARRAYS
37   REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
38   REAL (C_FLOAT), POINTER :: C_ARRAY(:)
39   ...
40   ! Associate C_ARRAY with an array allocated in C
41   CALL C_F_POINTER (ARRAYS%C, C_ARRAY, [ARRAYS%LENC])
42   ...

```

```

1      ! Allocate an array and make it available in C
2      ARRAYS%LENF = 100
3      ALLOCATE (ETA(ARRAYS%LENF))
4      ARRAYS%F = C_LOC(ETA)
5      ...
6  END SUBROUTINE SIMULATION

```

7 C Struct Declaration

```

8      struct pass {int lenc, lenf; float *c, *f;};

```

9 C Function Prototype:

```

10     void simulation(long alpha, double *beta, long *gamma,
11         double delta[], struct pass *arrays);

```

12 C Calling Sequence:

```

13         simulation(alpha, &beta, &gamma, delta, &arrays);

```

14 The above-listed Fortran code specifies a subroutine SIMULATION. This subroutine corresponds to the
15 C void function `simulation`.

16 The Fortran subroutine references the intrinsic module ISO_C_BINDING.

17 The first Fortran dummy argument of the subroutine is ALPHA, which has the type INTEGER(C_-
18 LONG) and the attribute VALUE. This dummy argument corresponds to the C formal parameter
19 `alpha`, which is a long. The actual C parameter is also a long.

20 The second Fortran dummy argument of the subroutine is BETA, which has the type REAL(C_-
21 DOUBLE) and the INTENT(INOUT) attribute. This dummy argument corresponds to the C formal
22 parameter `beta`, which is a pointer to double. An address is passed as the actual parameter in the C
23 calling sequence.

24 The third Fortran dummy argument of the subroutine is GAMMA, which has the type INTEGER(C_-
25 LONG) and the INTENT(OUT) attribute. This dummy argument corresponds to the C formal param-
26 eter `gamma`, which is a pointer to long. An address is passed as the actual parameter in the C calling
27 sequence.

28 The fourth Fortran dummy argument is the assumed-size array DELTA, which has the type REAL
29 (C_DOUBLE) and the attribute INTENT(IN). This dummy argument corresponds to the C formal
30 parameter `delta`, which is a double array. The actual C parameter is also a double array.

31 The fifth Fortran dummy argument is ARRAYS, which is a structure for accessing an array allocated
32 in C and an array allocated in Fortran. The lengths of these arrays are held in the components LENC
33 and LENF; their C addresses are held in components C and F.

34 C.11.2.3 Example of calling C functions with noninteroperable data

35 Many Fortran processors support 16-byte real numbers, which might not be supported by the C processor.
36 Assume a Fortran programmer wants to use a C procedure from a message passing library for an array
37 of these reals. The C prototype of this procedure is

1 void ProcessBuffer(void *buffer, int n_bytes);

2 with the corresponding Fortran interface

```

3 USE, INTRINSIC :: ISO_C_BINDING
4
5 INTERFACE
6     SUBROUTINE PROCESS_BUFFER(BUFFER,N_BYTES) BIND(C,NAME="ProcessBuffer")
7         IMPORT :: C_PTR, C_INT
8         TYPE(C_PTR), VALUE :: BUFFER ! The 'C address' of the array buffer
9         INTEGER(C_INT), VALUE :: N_BYTES ! Number of bytes in buffer
10    END SUBROUTINE PROCESS_BUFFER
11 END INTERFACE
```

12 This may be done using C_LOC if the particular Fortran processor specifies that C_LOC returns an
13 appropriate address:

```

14 REAL(R_QUAD), DIMENSION(:), ALLOCATABLE, TARGET :: QUAD_ARRAY
15 ...
16 CALL PROCESS_BUFFER(C_LOC(QUAD_ARRAY), INT(16*SIZE(QUAD_ARRAY),C_INT))
17 ! One quad real takes 16 bytes on this processor
```

18 C.11.2.4 Example of opaque communication between C and Fortran

19 The following example demonstrates how a Fortran processor can make a modern OO random number
20 generator written in Fortran available to a C program:

```

21 USE, INTRINSIC :: ISO_C_BINDING
22 ! Assume this code is inside a module
23
24 TYPE RANDOM_STREAM
25 ! A (uniform) random number generator (URNG)
26 CONTAINS
27     PROCEDURE(RANDOM_UNIFORM), DEFERRED, PASS(STREAM) :: NEXT
28     ! Generates the next number from the stream
29 END TYPE RANDOM_STREAM
30
31 ABSTRACT INTERFACE
32 ! Abstract interface of Fortran URNG
33 SUBROUTINE RANDOM_UNIFORM(STREAM, NUMBER)
34     IMPORT :: RANDOM_STREAM, C_DOUBLE
35     CLASS(RANDOM_STREAM), INTENT(INOUT) :: STREAM
36     REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
37 END SUBROUTINE RANDOM_UNIFORM
38 END INTERFACE
```

1 A polymorphic object of base type RANDOM_STREAM is not interoperable with C. However, we can
2 make such a random number generator available to C by packaging it inside another nonpolymorphic,
3 nonparameterized derived type:

```
4 TYPE :: URNG_STATE ! No BIND(C), as this type is not interoperable
5   CLASS(RANDOM_STREAM), ALLOCATABLE :: STREAM
6 END TYPE URNG_STATE
```

7 The following two procedures will enable a C program to use our Fortran uniform random number
8 generator:

```
9 ! Initialize a uniform random number generator:
10 SUBROUTINE INITIALIZE_URNG(STATE_HANDLE, METHOD) &
11     BIND(C, NAME="InitializeURNG")
12     TYPE(C_PTR), INTENT(OUT) :: STATE_HANDLE
13     ! An opaque handle for the URNG
14     CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: METHOD
15     ! The algorithm to be used
16
17     TYPE(URNG_STATE), POINTER :: STATE
18     ! An actual URNG object
19
20     ALLOCATE(STATE)
21     ! There needs to be a corresponding finalization
22     ! procedure to avoid memory leaks, not shown in this example
23     ! Allocate STATE%STREAM with a dynamic type depending on METHOD
24     ...
25     STATE_HANDLE=C_LOC(STATE)
26     ! Obtain an opaque handle to return to C
27 END SUBROUTINE INITIALIZE_URNG
28
29 ! Generate a random number:
30 SUBROUTINE GENERATE_UNIFORM(STATE_HANDLE, NUMBER) &
31     BIND(C, NAME="GenerateUniform")
32     TYPE(C_PTR), INTENT(IN), VALUE :: STATE_HANDLE
33     ! An opaque handle: Obtained via a call to INITIALIZE_URNG
34     REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
35
36     TYPE(URNG_STATE), POINTER :: STATE
37     ! A pointer to the actual URNG
38
39     CALL C_F_POINTER(CPTR=STATE_HANDLE, FPTR=STATE)
40     ! Convert the opaque handle into a usable pointer
41     CALL STATE%STREAM%NEXT(NUMBER)
42     ! Use the type-bound procedure NEXT to generate NUMBER
43 END SUBROUTINE GENERATE_UNIFORM
```

1 C.12 Clause 16 notes

2 C.12.1 Examples of host association (16.5.1.4)

3 The first two examples are examples of valid host association. The third example is an example of invalid
4 host association.

5 Example 1:

```
6 PROGRAM A
7     INTEGER I, J
8     ...
9 CONTAINS
10    SUBROUTINE B
11        INTEGER I ! Declaration of I hides
12                ! program A's declaration of I
13        ...
14        I = J      ! Use of variable J from program A
15                ! through host association
16    END SUBROUTINE B
17 END PROGRAM A
```

18 Example 2:

```
19 PROGRAM A
20     TYPE T
21     ...
22 END TYPE T
23 ...
24 CONTAINS
25    SUBROUTINE B
26        IMPLICIT TYPE (T) (C) ! Refers to type T declared below
27                                ! in subroutine B, not type T
28                                ! declared above in program A
29        ...
30        TYPE T
31        ...
32    END TYPE T
33    ...
34 END SUBROUTINE B
35 END PROGRAM A
```

36 Example 3:

```
37 PROGRAM Q
38     REAL (KIND = 1) :: C
```

```

1      ...
2  CONTAINS
3      SUBROUTINE R
4          REAL (KIND = KIND (C)) :: D   ! Invalid declaration
5                                          ! See below
6          REAL (KIND = 2) :: C
7      ...
8  END SUBROUTINE R
9  END PROGRAM Q

```

10 In the declaration of D in subroutine R, the use of C would refer to the declaration of C in subroutine
 11 R, not program Q. However, it is invalid because the declaration of C is required to occur before it is
 12 used in the declaration of D (7.1.12).

13 **C.12.2 Rules ensuring unambiguous generics (12.4.3.4.5)**

14 The rules in 12.4.3.4.5 are intended to ensure

- 15 • that it is possible to reference each specific procedure in the generic collection,
- 16 • that for any valid reference to the generic procedure, the determination of the specific procedure
 17 referenced is unambiguous, and
- 18 • that the determination of the specific procedure referenced can be made before execution of the
 19 program begins (during compilation).

20 Specific procedures are distinguished by fixed properties of their arguments, specifically type, kind type
 21 parameters, and rank. A valid reference to one procedure in a generic collection will differ from another
 22 because it has an argument that the other cannot accept, because it is missing an argument that the
 23 other requires, or because one of these fixed properties is different.

24 Although the declared type of a data entity is a fixed property, polymorphic variables allow for a
 25 limited degree of type mismatch between dummy arguments and actual arguments, so the requirement
 26 for distinguishing two dummy arguments is type incompatibility, not merely different types. (This is
 27 illustrated in the **BAD6** example later in this note.)

28 That same limited type mismatch means that two dummy arguments that are not type incompatible
 29 can be distinguished on the basis of the values of the kind type parameters they have in common; if one
 30 of them has a kind type parameter that the other does not, that is irrelevant in distinguishing them.

31 Rank is a fixed property, but some forms of array dummy arguments allow rank mismatches when a
 32 procedure is referenced by its specific name. In order to allow rank to always be usable in distinguishing
 33 generics, such rank mismatches are disallowed for those arguments when the procedure is referenced as
 34 part of a generic. Additionally, the fact that elemental procedures can accept array arguments is not
 35 taken into account when applying these rules, so apparent ambiguity between elemental and nonelemental
 36 procedures is possible; in such cases, the reference is interpreted as being to the nonelemental procedure.

37 For procedures referenced as operators or defined-assignment, syntactically distinguished arguments are
 38 mapped to specific positions in the argument list, so the rule for distinguishing such procedures is that
 39 it be possible to distinguish the arguments at one of the argument positions.

40 For user-defined derived-type input/output procedures, only the **dtv** argument corresponds to something
 41 explicitly written in the program, so it is the **dtv** that is required to be distinguished. Because **dtv**

1 arguments are required to be scalar, they cannot differ in rank. Thus this rule effectively involves only
2 type and kind type parameters.

3 For generic procedures identified by names, the rules are more complicated because optional arguments
4 may be omitted and because arguments may be specified either positionally or by name.

5 In the special case of type-bound procedures with passed-object dummy arguments, the passed-object
6 argument is syntactically distinguished in the reference, so rule (2) can be applied. The type of passed-
7 object arguments is constrained in ways that prevent passed-object arguments in the same scoping unit
8 from being type incompatible. Thus this rule effectively involves only kind type parameters and rank.

9 The primary means of distinguishing named generics is rule (3). The most common application of that
10 rule is a single argument satisfying both (3a) and (3b):

```
11         INTERFACE GOOD1
12             FUNCTION F1A(X)
13                 REAL :: F1A,X
14             END FUNCTION F1A
15             FUNCTION F1B(X)
16                 INTEGER :: F1B,X
17             END FUNCTION F1B
18         END INTERFACE GOOD1
```

19 Whether one writes `GOOD1(1.0)` or `GOOD1(X=1.0)`, the reference is to `F1A` because `F1B` would require an
20 integer argument whereas these references provide the real constant 1.0.

21 This example and those that follow are expressed using interface bodies, with type as the distinguishing
22 property. This was done to make it easier to write and describe the examples. The principles being
23 illustrated are equally applicable when the procedures get their explicit interfaces in some other way or
24 when kind type parameters or rank are the distinguishing property.

25 Another common variant is the argument that satisfies (3a) and (3b) by being required in one specific
26 and completely missing in the other:

```
27         INTERFACE GOOD2
28             FUNCTION F2A(X)
29                 REAL :: F2A,X
30             END FUNCTION F2A
31             FUNCTION F2B(X,Y)
32                 COMPLEX :: F2B
33                 REAL :: X,Y
34             END FUNCTION F2B
35         END INTERFACE GOOD2
```

36 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
37 `F2B`, because `F2A` has no argument in the second position or with the name `Y`. This approach is used as
38 an alternative to optional arguments when one wants a function to have different result type, kind type
39 parameters, or rank, depending on whether the argument is present. In many of the intrinsic functions,
40 the `DIM` argument works this way.

41 It is possible to construct cases where different arguments are used to distinguish positionally and by
42 name:

```

1      INTERFACE GOOD3
2          SUBROUTINE S3A(W,X,Y,Z)
3              REAL :: W,Y
4              INTEGER :: X,Z
5          END SUBROUTINE S3A
6          SUBROUTINE S3B(X,W,Z,Y)
7              REAL :: W,Z
8              INTEGER :: X,Y
9          END SUBROUTINE S3B
10     END INTERFACE GOOD3

```

11 If one writes `GOOD3(1.0,2,3.0,4)` to reference `S3A`, then the third and fourth arguments are consistent
 12 with a reference to `S3B`, but the first and second are not. If one switches to writing the first two
 13 arguments as keyword arguments in order for them to be consistent with a reference to `S3B`, the latter
 14 two arguments must also be written as keyword arguments, `GOOD3(X=2,W= 1.0,Z=4,Y=3.0)`, and the
 15 named arguments `Y` and `Z` are distinguished.

16 The ordering requirement in rule (3) is critical:

```

17     INTERFACE BAD4 ! this interface is invalid !
18         SUBROUTINE S4A(W,X,Y,Z)
19             REAL :: W,Y
20             INTEGER :: X,Z
21         END SUBROUTINE S4A
22         SUBROUTINE S4B(X,W,Z,Y)
23             REAL :: X,Y
24             INTEGER :: W,Z
25         END SUBROUTINE S4B
26     END INTERFACE BAD4

```

27 In this example, the positionally distinguished arguments are `Y` and `Z`, and it is `W` and `X` that are
 28 distinguished by name. In this order it is possible to write `BAD4(1.0,2,Y=3.0,Z=4)`, which is a valid
 29 reference for both `S4A` and `S4B`.

30 Rule (1) can be used to distinguish some cases that are not covered by rule (3):

```

31     INTERFACE GOOD5
32         SUBROUTINE S5A(X)
33             REAL :: X
34         END SUBROUTINE S5A
35         SUBROUTINE S5B(Y,X)
36             REAL :: Y,X
37         END SUBROUTINE S5B
38     END INTERFACE GOOD5

```

39 In attempting to apply rule (3), position 2 and name `Y` are distinguished, but they are in the wrong
 40 order, just like the `BAD4` example. However, when we try to construct a similarly ambiguous reference,
 41 we get `GOOD5(1.0,X=2.0)`, which can't be a reference to `S5A` because it would be attempting to associate

1 two different actual arguments with the dummy argument **X**. Rule (3) catches this case by recognizing
2 that **S5B** requires two real arguments, and **S5A** cannot possibly accept more than one.

3 The application of rule (1) becomes more complicated when extensible types are involved. If **FRUIT** is
4 an extensible type, **PEAR** and **APPLE** are extensions of **FRUIT**, and **BOSC** is an extension of **PEAR**, then

```
5      INTERFACE BAD6  ! this interface is invalid !
6          SUBROUTINE S6A(X,Y)
7              CLASS(PEAR) :: X,Y
8          END SUBROUTINE S6A
9          SUBROUTINE S6B(X,Y)
10             CLASS(FRUIT) :: X
11             CLASS(BOSC) :: Y
12          END SUBROUTINE S6B
13      END INTERFACE BAD6
```

14 might, at first glance, seem distinguishable this way, but because of the limited type mismatching allowed,
15 **BAD6(A_PEAR,A_BOSC)** is a valid reference to both **S6A** and **S6B**.

16 It is important to try rule (1) for each type present:

```
17      INTERFACE GOOD7
18          SUBROUTINE S7A(X,Y,Z)
19              CLASS(PEAR) :: X,Y,Z
20          END SUBROUTINE S7A
21          SUBROUTINE S7B(X,Z,W)
22              CLASS(FRUIT) :: X
23              CLASS(BOSC) :: Z
24              CLASS(APPLE),OPTIONAL :: W
25          END SUBROUTINE S7B
26      END INTERFACE GOOD7
```

27 Looking at the most general type, **S7A** has a minimum and maximum of 3 **FRUIT** arguments, while **S7B**
28 has a minimum of 2 and a maximum of three. Looking at the most specific, **S7A** has a minimum of 0
29 and a maximum of 3 **BOSC** arguments, while **S7B** has a minimum of 1 and a maximum of 2. However,
30 when we look at the intermediate, **S7A** has a minimum and maximum of 3 **PEAR** arguments, while **S7B**
31 has a minimum of 1 and a maximum of 2. Because **S7A**'s minimum exceeds **S7B**'s maximum, they can
32 be distinguished.

33 In identifying the minimum number of arguments with a particular set of properties, we exclude optional
34 arguments and test **TKR** compatibility, so the corresponding actual arguments are required to have
35 those properties. In identifying the maximum number of arguments with those properties, we include
36 the optional arguments and test not distinguishable, so we include actual arguments which could have
37 those properties but are not required to have them.

38 These rules are sufficient to ensure that references to procedures that meet them are unambiguous, but
39 there remain examples that fail to meet these rules but which can be shown to be unambiguous:

```
40      INTERFACE BAD8  ! this interface is invalid !
```

```

1      ! despite the fact that it is unambiguous !
2      SUBROUTINE S8A(X,Y,Z)
3          REAL,OPTIONAL :: X
4          INTEGER :: Y
5          REAL :: Z
6      END SUBROUTINE S8A
7      SUBROUTINE S8B(X,Z,Y)
8          INTEGER,OPTIONAL :: X
9          INTEGER :: Z
10         REAL :: Y
11     END SUBROUTINE S8B
12 END INTERFACE BAD8

```

13 This interface fails rule (3) because there are no required arguments that can be distinguished from the
14 positionally corresponding argument, but in order for the mismatch of the optional arguments not to
15 be relevant, the later arguments must be specified as keyword arguments, so distinguishing by name
16 does the trick. This interface is nevertheless invalid so a standard- conforming Fortran processor is not
17 required to do such reasoning. The rules to cover all cases are too complicated to be useful.

18 The real data objects that would be valid arguments for **S9A** are entirely disjoint from procedures that
19 are valid arguments to **S9B** and **S9C**, and the procedures that valid arguments for **S9B** are disjoint from
20 the procedures that are valid arguments to **S9C** because the former are required to accept real arguments
21 and the latter integer arguments. Again, this interface is invalid, so a standard-conforming Fortran
22 processor need not examine such properties when deciding whether a generic collection is valid. Again,
23 the rules to cover all cases are too complicated to be useful.

24 C.13 Array feature notes

25 C.13.1 Summary of features

26 This subclause is a summary of the principal array features.

27 C.13.1.1 Whole array expressions and assignments (7.2.1.2, 7.2.1.3)

28 An important feature is that whole array expressions and assignments are permitted. For example, the
29 statement

```
30 A = B + C * SIN (D)
```

31 where A, B, C, and D are arrays of the same shape, is permitted. It is interpreted element-by-element;
32 that is, the sine function is taken on each element of D, each result is multiplied by the corresponding
33 element of C, added to the corresponding element of B, and assigned to the corresponding element of
34 A. Functions, including user-written functions, may be arrays and may be generic with scalar versions.
35 All arrays in an expression or across an assignment shall conform; that is, have exactly the same shape
36 (number of dimensions and extents in each dimension), but scalars may be included freely and these are
37 interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment
38 takes place.

39 C.13.1.2 Array sections (2.4.5, 6.2.2.2)

40 Whenever whole arrays may be used, it is also possible to use subarrays called “sections”. For example:

1 A (:, 1:N, 2, 3:1:-1)

2 consists of a subarray containing the whole of the first dimension, positions 1 to N of the second dimension, position 2 of the third dimension and positions 1 to 3 in reverse order of the fourth dimension. This is an artificial example chosen to illustrate the different forms. Of course, a common use may be to select a row or column of an array, for example:

6 A (:, J)

7 C.13.1.3 WHERE statement (7.2.3)

8 The WHERE statement applies a conforming logical array as a mask on the individual operations in the expression and in the assignment. For example:

10 WHERE (A > 0) B = LOG (A)

11 takes the logarithm only for positive components of A and makes assignments only in these positions.

12 The WHERE statement also has a block form (WHERE construct).

13 C.13.1.4 Automatic arrays and allocatable variables (5.2, 5.3.7.4)

14 Two features useful for writing modular software are automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable variables, including arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

18 SUBROUTINE X (N, A, B)

19 REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)

20 specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will be needed for some allocatable variables.

23 C.13.1.5 Array constructors (4.7)

24 Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

25 [1.0, 3.0, 7.2]

26 which is a rank-one array of size 3,

27 [(1.3, 2.7, L = 1, 10), 7.1]

28 which is a rank-one array of size 21 and contains the pair of real constants 1.3 and 2.7 repeated 10 times followed by 7.1, and

30 [(I, I = 1, N)]

31 which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but higher dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

33 C.13.2 Examples

34 The array features have the potential to simplify the way that almost any array-using program is conceived and written. Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays.

1 C.13.2.1 Unconditional array computations

2 At the simplest level, statements such as

3 $A = B + C$

4 or

5 $S = \text{SUM} (A)$

6 can take the place of entire DO loops. The loops were required to perform array addition or to sum all
7 the elements of an array.

8 Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply	$P = \text{MATMUL} (Q, R)$
largest array element	$L = \text{MAXVAL} (P)$
factorial N	$F = \text{PRODUCT} ([(K, K = 2, N)])$

9 The Fourier sum $F = \sum_{i=1}^N a_i \times \cos x_i$ may also be computed without writing a DO loop if one makes
10 use of the element-by-element definition of array expressions as described in Clause 7. Thus, we can
11 write

12 $F = \text{SUM} (A * \text{COS} (X))$

13 The successive stages of calculation of F would then involve the arrays:

A	=	[A (1), ..., A (N)]
X	=	[X (1), ..., X (N)]
COS (X)	=	[COS (X (1)), ..., COS (X (N))]
A * COS (X)	=	[A (1) * COS (X (1)), ..., A (N) * COS (X (N))]

14 The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the
15 processor is dealing with arrays at every step of the calculation.

16 C.13.2.2 Conditional array computations

17 Suppose we wish to compute the Fourier sum in the above example, but to include only those terms
18 $a(i) \cos x(i)$ that satisfy the condition that the coefficient $a(i)$ is less than 0.01 in absolute value. More
19 precisely, we are now interested in evaluating the conditional Fourier sum

$$CF = \sum_{|a_i| < 0.01} a_i \times \cos x_i$$

20 where the index runs from 1 to N as before.

21 This can be done by using the MASK parameter of the SUM function, which restricts the summation
22 of the elements of the array $A * \text{COS} (X)$ to those elements that correspond to true elements of MASK.
23 Clearly, the mask required is the logical array expression $\text{ABS} (A) < 0.01$. Note that the stages of
24 evaluation of this expression are:

A	=	[A (1), ..., A (N)]
ABS (A)	=	[ABS (A (1)), ..., ABS (A (N))]
ABS (A) < 0.01	=	[ABS (A (1)) < 0.01, ..., ABS (A (N)) < 0.01]

1 The conditional Fourier sum we arrive at is:

2 `CF = SUM (A * COS (X), MASK = ABS (A) < 0.01)`

3 If the mask is all false, the value of CF is zero.

4 The use of a mask to define a subset of an array is crucial to the action of the WHERE statement. Thus
5 for example, to zero an entire array, we may write simply `A = 0`; but to set only the negative elements
6 to zero, we need to write the conditional assignment

7 `WHERE (A .LT. 0) A = 0`

8 The WHERE statement complements ordinary array assignment by providing array assignment to any
9 subset of an array that can be restricted by a logical expression.

10 In the Ising model described below, the WHERE statement predominates in use over the ordinary array
11 assignment statement.

12 **C.13.2.3 A simple program: the Ising model**

13 The Ising model is a well-known Monte Carlo simulation in 3-dimensional Euclidean space which is
14 useful in certain physical studies. We will consider in some detail how this might be programmed. The
15 model may be described in terms of a logical array of shape N by N by N. Each gridpoint is a single
16 logical variable which is to be interpreted as either an up-spin (true) or a down-spin (false).

17 The Ising model operates by passing through many successive states. The transition to the next state is
18 governed by a local probabilistic process. At each transition, all gridpoints change state simultaneously.
19 Every spin either flips to its opposite state or not according to a rule that depends only on the states
20 of its 6 nearest neighbors in the surrounding grid. The neighbors of gridpoints on the boundary faces of
21 the model cube are defined by assuming cubic periodicity. In effect, this extends the grid periodically
22 by replicating it in all directions throughout space.

23 The rule states that a spin is flipped to its opposite parity for certain gridpoints where a mere 3 or
24 fewer of the 6 nearest neighbors have the same parity as it does. Also, the flip is executed only with
25 probability P (4), P (5), or P (6) if as many as 4, 5, or 6 of them have the same parity as it does. (The
26 rule seems to promote neighborhood alignments that may presumably lead to equilibrium in the long
27 run.)

28 **C.13.2.3.1 Problems to be solved**

29 Some of the programming problems that we will need to solve in order to translate the Ising model into
30 Fortran statements using entire arrays are

- 31 (1) counting nearest neighbors that have the same spin,
- 32 (2) providing an array function to return an array of random numbers, and
- 33 (3) determining which gridpoints are to be flipped.

34 **C.13.2.3.2 Solutions in Fortran**

35 The arrays needed are:

36 `LOGICAL ISING (N, N, N), FLIPS (N, N, N)`

37 `INTEGER ONES (N, N, N), COUNT (N, N, N)`

38 `REAL THRESHOLD (N, N, N)`

The array function needed is:

```

1
2 FUNCTION RAND (N)
3 REAL RAND (N, N, N)

4 The transition probabilities are specified in the array

5 REAL P (6)

6 The first task is to count the number of nearest neighbors of each gridpoint g that have the same spin
7 as g.

8 Assuming that ISING is given to us, the statements

9 ONES = 0
10 WHERE (ISING) ONES = 1

11 make the array ONES into an exact analog of ISING in which 1 stands for an up-spin and 0 for a
12 down-spin.

13 The next array we construct, COUNT, will record for every gridpoint of ISING the number of spins to
14 be found among the 6 nearest neighbors of that gridpoint. COUNT will be computed by adding together
15 6 arrays, one for each of the 6 relative positions in which a nearest neighbor is found. Each of the 6
16 arrays is obtained from the ONES array by shifting the ONES array one place circularly along one of
17 its dimensions. This use of circular shifting imparts the cubic periodicity.

18 COUNT = CSHIFT (ONES, SHIFT = -1, DIM = 1) &
19         + CSHIFT (ONES, SHIFT = 1, DIM = 1) &
20         + CSHIFT (ONES, SHIFT = -1, DIM = 2) &
21         + CSHIFT (ONES, SHIFT = 1, DIM = 2) &
22         + CSHIFT (ONES, SHIFT = -1, DIM = 3) &
23         + CSHIFT (ONES, SHIFT = 1, DIM = 3)

24 At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints where
25 the Ising model has a down-spin. But we want a count of down-spins at those gridpoints, so we correct
26 COUNT at the down (false) points of ISING by writing:

27 WHERE (.NOT. ISING) COUNT = 6 - COUNT

28 Our object now is to use these counts of what may be called the "like-minded nearest neighbors" to
29 decide which gridpoints are to be flipped. This decision will be recorded as the true elements of an array
30 FLIPS. The decision to flip will be based on the use of uniformly distributed random numbers from the
31 interval  $0 \leq p < 1$ . These will be provided at each gridpoint by the array function RAND. The flip will
32 occur at a given point if and only if the random number at that point is less than a certain threshold
33 value. In particular, by making the threshold value equal to 1 at the points where there are 3 or fewer
34 like-minded nearest neighbors, we guarantee that a flip occurs at those points (because p is always less
35 than 1). Similarly, the threshold values corresponding to counts of 4, 5, and 6 are assigned P (4), P (5),
36 and P (6) in order to achieve the desired probabilities of a flip at those points (P (4), P (5), and P (6)
37 are input parameters in the range 0 to 1).

38 The thresholds are established by the statements:

39 THRESHOLD = 1.0
   WHERE (COUNT == 4) THRESHOLD = P (4)

```

```

1
2 WHERE (COUNT == 5) THRESHOLD = P (5)
3 WHERE (COUNT == 6) THRESHOLD = P (6)

```

4 and the spins that are to be flipped are located by the statement:

```

5 FLIPS = RAND (N) <= THRESHOLD

```

6 All that remains to complete one transition to the next state of the ISING model is to reverse the spins
7 in ISING wherever FLIPS is true:

```

8 WHERE (FLIPS) ISING = .NOT. ISING

```

9 **C.13.2.3.3 The complete Fortran subroutine**

10 The complete code, enclosed in a subroutine that performs a sequence of transitions, is as follows:

```

11 SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)
12
13     LOGICAL ISING (N, N, N), FLIPS (N, N, N)
14     INTEGER ONES (N, N, N), COUNT (N, N, N)
15     REAL THRESHOLD (N, N, N), P (6)
16
17     DO I = 1, ITERATIONS
18         ONES = 0
19         WHERE (ISING) ONES = 1
20         COUNT = CSHIFT (ONES, -1, 1) + CSHIFT (ONES, 1, 1) &
21             + CSHIFT (ONES, -1, 2) + CSHIFT (ONES, 1, 2) &
22             + CSHIFT (ONES, -1, 3) + CSHIFT (ONES, 1, 3)
23         WHERE (.NOT. ISING) COUNT = 6 - COUNT
24         THRESHOLD = 1.0
25         WHERE (COUNT == 4) THRESHOLD = P (4)
26         WHERE (COUNT == 5) THRESHOLD = P (5)
27         WHERE (COUNT == 6) THRESHOLD = P (6)
28         FLIPS = RAND (N) <= THRESHOLD
29         WHERE (FLIPS) ISING = .NOT. ISING
30     END DO
31
32 CONTAINS
33     FUNCTION RAND (N)
34         REAL RAND (N, N, N)
35         CALL RANDOM_NUMBER (HARVEST = RAND)
36         RETURN
37     END FUNCTION RAND
38 END

```

39 **C.13.2.3.4 Reduction of storage**

1 The array ISING could be removed (at some loss of clarity) by representing the model in ONES all the
 2 time. The array FLIPS can be avoided by combining the two statements that use it as:

3 WHERE (RAND (N) <= THRESHOLD) ISING = .NOT. ISING

4 but an extra temporary array would probably be needed. Thus, the scope for saving storage while
 5 performing whole array operations is limited. If N is small, this will not matter and the use of whole
 6 array operations is likely to lead to good execution speed. If N is large, storage may be very important
 7 and adequate efficiency will probably be available by performing the operations plane by plane. The
 8 resulting code is not as elegant, but all the arrays except ISING will have size of order N^2 instead of N^3 .

9 **C.13.3 FORMula TRANslation and array processing**

10 Many mathematical formulas can be translated directly into Fortran by use of the array processing
 11 features.

12 We assume the following array declarations:

13 REAL X (N), A (M, N)

14 Some examples of mathematical formulas and corresponding Fortran expressions follow.

15 **C.13.3.1 A sum of products**

The expression

$$\sum_{j=1}^N \prod_{i=1}^M a_{ij}$$

16 can be formed using the Fortran expression

17 SUM (PRODUCT (A, DIM=1))

18 The argument DIM=1 means that the product is to be computed down each column of A. If A had the
 19 value $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$ the result of this expression is $BE + CF + DG$.

20 **C.13.3.2 A product of sums**

The expression

$$\prod_{i=1}^M \sum_{j=1}^N a_{ij}$$

21 can be formed using the Fortran expression

22 PRODUCT (SUM (A, DIM = 2))

23 The argument DIM = 2 means that the sum is to be computed along each row of A. If A had the
 24 value $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$ the result of this expression is $(B+C+D)(E+F+G)$.

25 **C.13.3.3 Addition of selected elements**

The expression

$$\sum_{x_i > 0.0} x_i$$

1 can be formed using the Fortran expression

2 `SUM (X, MASK = X > 0.0)`

3 The mask locates the positive elements of the array of rank one. If X has the vector value (0.0, -0.1,
4 0.2, 0.3, 0.2, -0.1, 0.0), the result of this expression is 0.7.

5 **C.13.4 Sum of squared residuals**

The expression

$$\sum_{i=1}^N (x_i - x_{\text{mean}})^2$$

6 can be formed using the Fortran statements

7 `XMEAN = SUM (X) / SIZE (X)`

8 `SS = SUM ((X - XMEAN) ** 2)`

9 Thus, SS is the sum of the squared residuals.

10 **C.13.5 Vector norms: infinity-norm and one-norm**

11 The infinity-norm of vector $X = (X(1), \dots, X(N))$ is defined as the largest of the numbers $ABS(X(1)),$
12 $\dots, ABS(X(N))$ and therefore has the value `MAXVAL (ABS(X))`.

13 The one-norm of vector X is defined as the *sum* of the numbers $ABS(X(1)), \dots, ABS(X(N))$ and
14 therefore has the value `SUM (ABS(X))`.

15 **C.13.6 Matrix norms: infinity-norm and one-norm**

16 The infinity-norm of the matrix $A = (A(I, J))$ is the largest row-sum of the matrix $ABS(A(I, J))$ and
17 therefore has the value `MAXVAL (SUM (ABS(A), DIM = 2))`.

18 The one-norm of the matrix $A = (A(I, J))$ is the largest column-sum of the matrix $ABS(A(I, J))$ and
19 therefore has the value `MAXVAL (SUM (ABS(A), DIM = 1))`.

20 **C.13.7 Logical queries**

21 The intrinsic functions allow quite complicated questions about tabular data to be answered without
22 use of loops or conditional constructs. Consider, for example, the questions asked below about a simple
23 tabulation of students' test scores.

24 Suppose the rectangular table T (M, N) contains the test scores of M students who have taken N different
25 tests. T is an integer matrix with entries in the range 0 to 100.

26 Example: The scores on 4 tests made by 3 students are held as the table

T =

$$\begin{bmatrix} 85 & 76 & 90 & 60 \\ 71 & 45 & 50 & 80 \\ 66 & 45 & 21 & 55 \end{bmatrix}$$

27 Question: What is each student's top score?

28 Answer: `MAXVAL (T, DIM = 2)`; in the example: [90, 80, 66].

1 Question: What is the average of all the scores?

2 Answer: $\text{SUM (T) / SIZE (T)}$; in the example: 62.

3 Question: How many of the scores in the table are above average?

4 Answer: $\text{ABOVE} = \text{T} > \text{SUM (T) / SIZE (T)}$; $\text{N} = \text{COUNT (ABOVE)}$; in the example: ABOVE is the
 5 logical array ($t = \text{true}$, $. = \text{false}$): $\begin{bmatrix} t & t & t & . \\ t & . & . & t \\ t & . & . & . \end{bmatrix}$ and COUNT (ABOVE) is 6.

6 Question: What was the lowest score in the above-average group of scores?

7 Answer: $\text{MINVAL (T, MASK = ABOVE)}$, where ABOVE is as defined previously; in the example: 66.

8 Question: Was there a student whose scores were all above average?

9 Answer: With ABOVE as previously defined, the answer is yes or no according as the value of the
 10 expression $\text{ANY (ALL (ABOVE, DIM = 2))}$ is true or false; in the example, the answer is no.

11 C.13.8 Parallel computations

12 The most straightforward kind of parallel processing is to do the same thing at the same time to many
 13 operands. Matrix addition is a good example of this very simple form of parallel processing. Thus, the
 14 array assignment $A = B + C$ specifies that corresponding elements of the identically-shaped arrays B
 15 and C be added together in parallel and that the resulting sums be assigned in parallel to the array A.

16 The process being done in parallel in the example of matrix addition is of course the process of addi-
 17 tion; the array feature that implements matrix addition as a parallel process is the element-by-element
 18 evaluation of array expressions.

19 These observations lead us to look to element-by-element computation as a means of implementing other
 20 simple parallel processing algorithms.

21 C.13.9 Example of element-by-element computation

22 Several polynomials of the same degree may be evaluated at the same point by arranging their coefficients
 23 as the rows of a matrix and applying Horner's method for polynomial evaluation to the columns of the
 24 matrix so formed.

25 The procedure is illustrated by the code to evaluate the three cubic polynomials

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$

$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$

$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

26 in parallel at the point $t = X$ and to place the resulting vector of numbers $[P(X), Q(X), R(X)]$ in the
 27 real array RESULT (3).

28 The code to compute RESULT is just the one statement

29 $\text{RESULT} = \text{M}(:, 1) + X * (\text{M}(:, 2) + X * (\text{M}(:, 3) + X * \text{M}(:, 4)))$

where M represents the matrix M (3, 4) with value $\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$.

1 C.13.10 Bit manipulation and inquiry procedures

- 2 The procedures IOR, IAND, NOT, IEOR, ISHFT, ISHFTC, IBITS, MVBITS, BTEST, IBSET, and
3 IBCLR are defined by MIL-STD 1753 for scalar arguments and are extended in this part of ISO/IEC
4 1539 to accept array arguments and to return array results.

Annex D

(Informative)

Processor Dependencies

D.1 Unspecified Items

This part of ISO/IEC 1539 does not specify the following properties:

- the properties listed in 1.4;
- a processor's error detection capabilities beyond those listed in 1.5;
- which additional intrinsic procedures or modules a processor provides (1.5);
- the number and kind of companion processors (2.5.10).

D.2 Processor Dependencies

According to this part of ISO/IEC 1539, the following properties are processor dependent:

- whether an external file is available to all images or only to particular images (2.3.2);
- the order of evaluation of the specification expressions within the specification part of an invoked Fortran procedure (2.3.6);
- the mechanism of a companion processor, and the means of selecting between multiple companion processors (2.5.10);
- the processor character set (3.1);
- the means for specifying the source form of a program unit (3.3);
- the maximum number of characters allowed on a source line containing characters not of default kind (3.3.1, 3.3.2);
- the maximum depth of nesting of include files (3.4);
- the interpretation of the *char-literal-constant* in the include line (3.4);
- whether comments in a macro definition appear in the expansion, whether continuations and consecutive blanks that are not part of a token are preserved by macro expansion, and the maximum number of characters produced by a macro expansion in a statement containing any character that is not of default kind (3.5.2.1);
- the set of values supported by an intrinsic type, other than logical and bits (4.1.1);
- the kind of a character length type parameter (4.4.5.1);
- whether particular control characters may appear within a character literal constant in fixed source form (4.4.5.3);
- the collating sequence for each character set (4.4.5.4);

- the upper limit of the size supported by the BITS type (4.4.7);
- the order of finalization when several objects are finalized as the consequence of a single event (4.5.6.2);
- whether an array is contiguous, except as specified in 5.3.6;
- the positive integer values assigned to the *stat-variable* in a STAT= specifier as the result of an error condition (6.3.4, 8.5.7);
- the value assigned to the *errmsg-variable* in an ERRMSG= specifier as the result of an error condition (6.3.5, 8.5.7);
- the kind of a numeric intrinsic binary operation where
 - (1) both operands are integer but with different kind type parameters, and the decimal exponent ranges are the same,
 - (2) both operands are any of type real or complex but with different kind type parameters, and the decimal precisions are the same,
 and for a logical intrinsic binary operation where the operands have different kind type parameters (7.1.9.2);
- the order of evaluation of the specification expressions within the specification part of a BLOCK construct when the construct is executed (8.1.4);
- the pointer association status of a pointer that has its pointer association changed in more than one iteration of a DO construct, on termination of the construct (8.1.7);
- the manner in which the stop code of a STOP statement is made available (8.4);
- the relationship between the file storage units when viewing a file as a stream file, and the records when viewing that file as a record file (9);
- whether particular control characters may appear in a formatted record or a formatted stream file (9.1.1);
- at any time, the set of allowed access methods, set of allowed forms, set of allowed actions, and set of allowed record lengths for a file (9.2);
- the set of allowable names for a file (9.2);
- the relationship between positions of successive file storage units in an external file that is connected for formatted stream access (9.2.2.3);
- the external unit preconnected for sequential formatted input and identified by an asterisk or the named constant INPUT_UNIT of the ISO_FORTRAN_ENV intrinsic module (9.4);
- the external unit preconnected for sequential formatted output and identified by an asterisk or the named constant OUTPUT_UNIT of the ISO_FORTRAN_ENV intrinsic module (9.4);
- the external unit preconnected for sequential formatted output and identified by the named constant ERROR_UNIT of the ISO_FORTRAN_ENV intrinsic module, and whether this unit is the same as OUTPUT_UNIT (9.4);
- at any time, the set of external units that exist for a program (9.4.2);
- whether a unit can be connected to a file that is also connected to a C stream (9.4.3);
- whether the files connected to the units INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT correspond to the predefined C text streams standard input, standard output, and standard error, respectively (9.4.3);

- the results of performing input/output operations on a file that is also connected to a C stream (9.4.3);
- the results of performing input/output operations on an external file both from Fortran and from a procedure defined by means other than Fortran (9.4.3);
- the default value for the ACTION= specifier on the OPEN statement (9.4.5.2);
- the encoding of a file opened with ENCODING='DEFAULT' (9.4.5.7);
- the file connected to by an OPEN statement with STATUS='SCRATCH' (9.4.5.8);
- the interpretation of case in a file name (9.4.5.8, 9.9.2.1);
- the default value for the RECL= specified in an OPEN statement (9.4.5.13);
- the effect of RECL= on a record containing any nondefault characters (9.4.5.13);
- the default I/O rounding mode (9.4.5.14);
- the default sign mode (9.4.5.15);
- the file status when STATUS='UNKNOWN' is specified in an OPEN statement (9.4.5.16);
- whether POS= is permitted with particular files, and whether POS= can position a particular file to a position prior to its current position (9.5.2.11);
- the result of unformatted input when the value stored in the file has a different type or type parameters from the input list item, as described in 9.5.4.4.1;
- the positive integer values assigned to the variable in an IOSTAT= specifier as the result of an error condition (9.10.4);
- the value assigned to the variable in an IOMSG= specifier as the result of an error condition (9.10.5);
- the variable in a POSITION= specifier in an INQUIRE statement as described in 9.9.2.22;
- the relationship between file size and the data stored in records in a sequential or direct access file (9.9.2.29);
- the number of file storage units needed to store data in an unformatted file (9.9.3);
- the set of error conditions that can occur in input/output statements (9.10);
- the result of output of non-representable characters to a Unicode file (10.7.1);
- the interpretation of the optional non-blank characters within the parentheses of a real NaN input field (10.7.2.3.2);
- the interpretation of a sign in a NaN input field (10.7.2.3.2);
- for output of an IEEE NaN, whether after the letters 'NaN', the processor produces additional alphanumeric characters enclosed in parentheses (10.7.2.3.2);
- the effect of the I/O rounding mode PROCESSOR_DEFINED (10.7.2.3.7);
- with an I/O rounding mode of NEAREST, for converting a value that is exactly halfway between the two nearest representable values in the result format, which value is chosen (10.7.2.3.7);
- the field width used for the B0, O0, and Z0 edit descriptors (10.7.2.4);
- the field width used for the G0 edit descriptor (10.7.5);

- the file position when position editing skips a character of type nondefault character in an external unit that is not connected to a Unicode file (10.8.1);
- when the sign mode is `PROCESSOR_DEFINED`, whether a plus sign appears in a numeric output field for a nonnegative value (10.8.4);
- the results of list-directed output, as described in 10.10.4;
- the results of namelist output, as described in 10.11.4;
- the interaction between argument association and pointer association, as described in 12.5.2.5;
- the values returned by some intrinsic functions, as described in 13;
- the extent to which a processor supports IEEE arithmetic (14);
- the values of the floating-point exception flags on entry to a procedure defined by means other than Fortran (15.5.3).

D.3 Unresolved Technical Issues

J3 internal note

Unresolved Technical Issue 100

Ungrammatical gobbledygook with nonsensical references.

That's the following paragraph.

The representation methods (4.1.1 Set of values) and the kind type parameter values specifying a representation method supported by a processor is not specified by this part of ISO/IEC 1539. (4.2 Type parameters).

It is also false to fact; the standard *does* specify the kind values for BITS.
When rewording,

- pay attention to singular and plural,
- only make true statements,
- find the normative text supporting the statements, and
- get the references to said text right.

J3 internal note

Unresolved Technical Issue 107

Not defined by Fortran means exactly that.

The text said:

“The interface of a procedure defined by means other than Fortran may be specified by an interface body or procedure declaration statement. The means by which such a procedure is defined are processor dependent (12.6.3).”

This is nonsense. If it is defined by means other than Fortran, it is not covered by the standard at all ... any more than your cup of coffee is. The means are not processor dependent, they are completely outwith the scope of the standard.

The standard is about the Fortran programming language. If you call a procedure not defined by Fortran it is *not defined by Fortran*. It is not “defined by Fortran to be processor-dependent”, it is just “not defined by Fortran”. End of story.

Fix the text in c12 to avoid saying this nonsense.

J3 internal note

Unresolved Technical Issue 108

Ordering requirement text not normative.

The quoted text here is not normative, and not supported by any normative text.

In particular, you cannot “give permission to do something” in a note.

If this is covered by the first exclusion, the note should be rewritten to reference that exclusion, and the paragraph above should be struck.

Annex E

(Informative)

Syntax rules

E.1 Extract of all syntax rules

Clause 1:

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

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

R103 *scalar-xyz* is *xyz*

C101 (R103) *scalar-xyz* shall be scalar.

Clause 2:

[illegible]

R202	<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>

[illegible][illegible][illegible]

R206	<i>implicit-part-stmt</i>	is	<i>implicit-stmt</i>
		or	<i>parameter-stmt</i>
		or	<i>format-stmt</i>
		or	<i>entry-stmt</i>

R207	<i>declaration-construct</i>	is <i>derived-type-def</i>
		or <i>entry-stmt</i>
		or <i>enum-def</i>
		or <i>format-stmt</i>
		or <i>interface-block</i>
		or <i>macro-definition</i>
		or <i>parameter-stmt</i>
		or <i>procedure-declaration-stmt</i>
		or <i>specification-stmt</i>
		or <i>type-declaration-stmt</i>
		or <i>stmt-function-stmt</i>

R208	<i>execution-part</i>	is <i>executable-construct</i> [<i>execution-part-construct</i>] ...
R209	<i>execution-part-construct</i>	is <i>executable-construct</i> or <i>format-stmt</i> or <i>entry-stmt</i> or <i>data-stmt</i>
R210	<i>internal-subprogram-part</i>	is <i>contains-stmt</i> [<i>internal-subprogram</i>] ...
R211	<i>internal-subprogram</i>	is <i>function-subprogram</i> or <i>subroutine-subprogram</i>
R212	<i>specification-stmt</i>	is <i>access-stmt</i> or <i>allocatable-stmt</i> or <i>asynchronous-stmt</i> or <i>bind-stmt</i> or <i>common-stmt</i> or <i>data-stmt</i> or <i>dimension-stmt</i> or <i>equivalence-stmt</i> or <i>external-stmt</i> or <i>intent-stmt</i> or <i>intrinsic-stmt</i> or <i>namelist-stmt</i> or <i>optional-stmt</i> or <i>pointer-stmt</i> or <i>protected-stmt</i> or <i>save-stmt</i> or <i>target-stmt</i> or <i>volatile-stmt</i> or <i>value-stmt</i>
R213	<i>executable-construct</i>	is <i>action-stmt</i> or <i>associate-construct</i> or <i>block-construct</i> or <i>case-construct</i> or <i>critical-construct</i> or <i>do-construct</i> or <i>forall-construct</i> or <i>if-construct</i> or <i>select-type-construct</i> or <i>where-construct</i>
R214	<i>action-stmt</i>	is <i>allocate-stmt</i> or <i>assignment-stmt</i> or <i>backspace-stmt</i> or <i>call-stmt</i> or <i>close-stmt</i> or <i>continue-stmt</i> or <i>cycle-stmt</i>

or *deallocate-stmt*
or *end-function-stmt*
or *end-mp-subprogram-stmt*
or *end-program-stmt*
or *end-subroutine-stmt*
or *endfile-stmt*
or *exit-stmt*
or *flush-stmt*
or *forall-stmt*
or *goto-stmt*
or *if-stmt*
or *inquire-stmt*
or *notify-stmt*
or *nullify-stmt*
or *open-stmt*
or *pointer-assignment-stmt*
or *print-stmt*
or *query-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 *wait-stmt*
or *where-stmt*
or *write-stmt*
or *arithmetic-if-stmt*
or *computed-goto-stmt*

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

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

Clause 3:

R301 *character* **is** *alphanumeric-character*
or *special-character*

R302 *alphanumeric-character* **is** *letter*
or *digit*
or *underscore*

R303 *underscore* **is** *-*

R304 *name* **is** *letter* [*alphanumeric-character*] ...

C301 (R304) The maximum length of a *name* is 63 characters.

R305 *constant* **is** *literal-constant*
or *named-constant*

R306 *literal-constant* **is** *int-literal-constant*

		or <i>real-literal-constant</i>
		or <i>complex-literal-constant</i>
		or <i>logical-literal-constant</i>
		or <i>char-literal-constant</i>
		or <i>bits-literal-constant</i>
R307	<i>named-constant</i>	is <i>name</i>
R308	<i>int-constant</i>	is <i>constant</i>
C302	(R308) <i>int-constant</i> shall be of type integer.	
R309	<i>char-constant</i>	is <i>constant</i>
C303	(R309) <i>char-constant</i> shall be of type character.	
R310	<i>intrinsic-operator</i>	is <i>power-op</i>
		or <i>mult-op</i>
		or <i>add-op</i>
		or <i>concat-op</i>
		or <i>rel-op</i>
		or <i>not-op</i>
		or <i>and-op</i>
		or <i>or-op</i>
		or <i>equiv-op</i>
R311	<i>defined-operator</i>	is <i>defined-unary-op</i>
		or <i>defined-binary-op</i>
		or <i>extended-intrinsic-op</i>
R312	<i>extended-intrinsic-op</i>	is <i>intrinsic-operator</i>
R313	<i>label</i>	is <i>digit</i> [<i>digit</i> [<i>digit</i> [<i>digit</i> [<i>digit</i>]]]]
C304	(R313) At least one digit in a <i>label</i> shall be nonzero.	
R314	<i>macro-definition</i>	is <i>define-macro-stmt</i>
		[<i>macro-declaration-stmt</i>] ...
		<i>macro-body-block</i>
		<i>end-macro-stmt</i>
R315	<i>define-macro-stmt</i>	is DEFINE MACRO [, <i>macro-attr-list</i>] :: <i>macro-name</i> ■ ■ [([<i>macro-dummy-arg-name-list</i>])]
C305	(R315) A <i>macro-dummy-arg-name</i> shall not appear more than once in a <i>macro-dummy-arg-name-list</i> .	
R316	<i>macro-attr</i>	is <i>access-spec</i>
R317	<i>macro-declaration-stmt</i>	is <i>macro-type-declaration-stmt</i>
		or <i>macro-optional-decl-stmt</i>
		or <i>macro-variable-decl-stmt</i>
R318	<i>macro-type-declaration-stmt</i>	is MACRO <i>macro-type-spec</i> :: <i>macro-local-variable-name-list</i>
R319	<i>macro-optional-decl-stmt</i>	is MACRO OPTIONAL :: <i>macro-dummy-arg-name-list</i>
R320	<i>macro-variable-decl-stmt</i>	is MACRO VARIABLE :: <i>macro-local-variable-name-list</i>
R321	<i>macro-type-spec</i>	is INTEGER [([<i>KIND=</i>] <i>macro-expr</i>)]
C306	(R318, <i>macro-variable-decl-stmt</i>) A <i>macro-local-variable-name</i> shall not be the same as the name of a dummy argument of the macro being defined.	
C307	(R319) A <i>macro-dummy-arg-name</i> shall be the name of a dummy argument of the macro being defined.	
C308	(R321) If <i>macro-expr</i> appears, when the macro is expanded <i>macro-expr</i> shall be of type integer, and have a non-negative value that specifies a representation method that exists on the processor.	

- R322 *macro-body-block* is [*macro-body-construct*] ...
- R323 *macro-body-construct* is *macro-definition*
or *expand-stmt*
or *macro-body-stmt*
or *macro-do-construct*
or *macro-if-construct*
or *macro-int-assignment-stmt*
or *macro-tok-assignment-stmt*
- C309 A statement in a macro definition that is not a *macro-body-construct* or *macro-definition* shall not appear on a line with any other statement.
- R324 *macro-do-construct* is *macro-do-stmt*
macro-body-block
macro-end-do-stmt
- R325 *macro-do-stmt* is MACRO DO *macro-do-variable-name* = *macro-do-limit* , ■
■ *macro-do-limit* [, *macro-do-limit*]
- C310 (R325) A *macro-do-variable-name* shall be a local variable of the macro being defined, and shall not be a macro dummy argument.
- R326 *macro-do-limit* is *macro-expr*
- C311 (R326) A *macro-do-limit* shall expand to an expression of type integer.
- R327 *macro-end-do-stmt* is MACRO END DO
- R328 *macro-if-construct* is *macro-if-then-stmt*
macro-body-block
[*macro-else-if-stmt*
macro-body-block] ...
[*macro-else-stmt*
macro-body-block]
macro-end-if-stmt
- R329 *macro-if-then-stmt* is MACRO IF (*macro-condition*) THEN
- R330 *macro-else-if-stmt* is MACRO ELSE IF (*macro-condition*) THEN
- R331 *macro-else-stmt* is MACRO ELSE
- R332 *macro-end-if-stmt* is MACRO END IF
- R333 *macro-condition* is *macro-expr*
- C312 (R333) A macro condition shall expand to an expression of type logical.
- R334 *macro-int-assignment-stmt* is MACRO *macro-integer-variable-name* = *macro-expr*
- C313 (R334) *macro-integer-variable-name* shall be the name of a macro local variable of type integer.
- R335 *macro-tok-assignment-stmt* is MACRO *macro-tok-variable-name* = *assignment-tok-sequence*
- C314 (R335) *macro-tok-variable-name* shall be the name of an untyped macro local variable that is not a macro dummy argument.
- R336 *assignment-tok-sequence* is *result-token* [*result-token*] ... [&&]
- R337 *macro-body-stmt* is *result-token* [*result-token*] ... [&&]
- C315 (R337) If the first *result-token* is MACRO the second *result-token* shall not be a keyword or name.
- C316 (R337) If the first token is DEFINE the second token shall not be MACRO.
- R338 *result-token* is *token* [%% *token*] ...
- R339 *token* is any lexical token including labels, keywords, and semi-colon.
- C317 && shall not appear in the last *macro-body-stmt* of a macro definition.
- C318 When a macro is expanded, the last *macro-body-stmt* processed shall not end with &&.

- R340 *end-macro-stmt* is END MACRO [*macro-name*]
- C319 (R314) The *macro-name* in the END MACRO statement shall be the same as the *macro-name* in the DEFINE MACRO statement.
- R341 *macro-expr* is *basic-token-sequence*
- C320 (R341) A *macro-expr* shall expand to a scalar initialization expression.
- R342 *expand-stmt* is EXPAND *macro-name* [(*macro-actual-arg-list*)]
- C321 (R342) *macro-name* shall be the name of a macro that was previously defined or accessed via use or host association.
- C322 (R342) The macro shall expand to a sequence or zero or more complete Fortran statements.
- C323 (R342) The statements produced by a macro expansion shall conform to the syntax rules and constraints as if they replaced the EXPAND statement prior to program processing.
- C324 (R342) The statements produced by a macro expansion shall not include a statement which ends the scoping unit containing the EXPAND statement.
- C325 (R342) If a macro expansion produces a statement which begins a new scoping unit, it shall also produce a statement which ends that scoping unit.
- C326 (R342) If the EXPAND statement appears as the *action-stmt* of an *if-stmt*, it shall expand to exactly one *action-stmt* that is not an *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, or *if-stmt*.
- C327 (R342) If the EXPAND statement appears as a *do-term-action-stmt*, it shall expand to exactly one *action-stmt* that is not an *arithmetic-if-stmt*, *continue-stmt*, *cycle-stmt*, *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, *exit-stmt*, *goto-stmt*, *return-stmt*, or *stop-stmt*.
- C328 (R342) If the EXPAND statement has a label, the expansion of the macro shall produce at least one statement, and the first statement produced shall not have a label.
- C329 (R342) A *macro-actual-arg* shall appear corresponding to each nonoptional macro dummy argument.
- C330 (R342) At most one *macro-actual-arg* shall appear corresponding to each optional macro dummy argument.
- R343 *macro-actual-arg* is [*macro-dummy-name* =] *macro-actual-arg-value*
- C331 (R343) *macro-dummy-name* shall be the name of a macro dummy argument of the macro being expanded.
- C332 (R342) The *macro-dummy-name*= shall not be omitted unless it has been omitted from each preceding *macro-actual-arg* in the *expand-stmt*.
- R344 *macro-actual-arg-value* is *basic-token-sequence*
- R345 *basic-token-sequence* is *basic-token*
or [*basic-token-sequence*] *nested-token-sequence* ■
■ [*basic-token-sequence*]
or *basic-token basic-token-sequence*
- R346 *basic-token* is any lexical token except comma, parentheses, array ■
■ constructor delimiters, and semi-colon.
- R347 *nested-token-sequence* is ([*arg-token*] ...)
or (/ [*arg-token*] ... /)
or lbracket [*arg-token*] ... rbracket
- R348 *arg-token* is *basic-token*
or ,
- C333 (R338) The result of token concatenation shall have the form of a lexical token.

Clause 4:

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

- or :**
- C401 (R401) The *type-param-value* for a kind type parameter shall be an initialization expression.
- C402 (R401) A colon shall not be used as a *type-param-value* except in the declaration of an entity or component that has the POINTER or ALLOCATABLE attribute.
- R402 *type-spec* **is** *intrinsic-type-spec*
 or *derived-type-spec*
- C403 (R402) The *derived-type-spec* shall not specify an abstract type (4.5.7).
- R403 *declaration-type-spec* **is** *intrinsic-type-spec*
 or TYPE (*intrinsic-type-spec*)
 or TYPE (*derived-type-spec*)
 or CLASS (*derived-type-spec*)
 or CLASS (*)
- C404 (R403) In a *declaration-type-spec*, every *type-param-value* that is not a colon or an asterisk shall be a *specification-expr*.
- C405 (R403) In a *declaration-type-spec* that uses the CLASS keyword, *derived-type-spec* shall specify an extensible type (4.5.7).
- C406 (R403) The TYPE(*derived-type-spec*) shall not specify an abstract type (4.5.7).
- C407 An entity declared with the CLASS keyword shall be a dummy argument or have the ALLOCATABLE or POINTER attribute. It shall not have the VALUE attribute.
- R404 *intrinsic-type-spec* **is** INTEGER [*kind-selector*]
 or REAL [*kind-selector*]
 or DOUBLE PRECISION
 or COMPLEX [*kind-selector*]
 or CHARACTER [*char-selector*]
 or LOGICAL [*kind-selector*]
 or BITS [*kind-selector*]
- R405 *kind-selector* **is** ([KIND =] *scalar-int-initialization-expr*)
- C408 (R405) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.
- R406 *signed-int-literal-constant* **is** [*sign*] *int-literal-constant*
- R407 *int-literal-constant* **is** *digit-string* [- *kind-param*]
- R408 *kind-param* **is** *digit-string*
 or *scalar-int-constant-name*
- R409 *signed-digit-string* **is** [*sign*] *digit-string*
- R410 *digit-string* **is** *digit* [*digit*] ...
- R411 *sign* **is** +
 or -
- C409 (R408) A *scalar-int-constant-name* shall be a named constant of type integer.
- C410 (R408) The value of *kind-param* shall be nonnegative.
- C411 (R407) The value of *kind-param* shall specify a representation method that exists on the processor.
- R412 *signed-real-literal-constant* **is** [*sign*] *real-literal-constant*
- R413 *real-literal-constant* **is** *significand* [*exponent-letter* *exponent*] [- *kind-param*]
 or *digit-string* *exponent-letter* *exponent* [- *kind-param*]
- R414 *significand* **is** *digit-string* . [*digit-string*]
 or . *digit-string*
- R415 *exponent-letter* **is** E

- or D
- R416 *exponent* is *signed-digit-string*
- C412 (R413) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.
- C413 (R413) The value of *kind-param* shall specify an approximation method that exists on the processor.
- R417 *complex-literal-constant* is (*real-part* , *imag-part*)
- R418 *real-part* is *signed-int-literal-constant*
or *signed-real-literal-constant*
or *named-constant*
- R419 *imag-part* is *signed-int-literal-constant*
or *signed-real-literal-constant*
or *named-constant*
- C414 (R417) Each named constant in a complex literal constant shall be of type integer or real.
- R420 *char-selector* is *length-selector*
or (LEN = *type-param-value* , ■
■ KIND = *scalar-int-initialization-expr*)
or (*type-param-value* , ■
■ [KIND =] *scalar-int-initialization-expr*)
or (KIND = *scalar-int-initialization-expr* ■
■ [, LEN = *type-param-value*])
- R421 *length-selector* is ([LEN =] *type-param-value*)
or * *char-length* [,]
- R422 *char-length* is (*type-param-value*)
or *int-literal-constant*
- C415 (R420) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.
- C416 (R422) The *int-literal-constant* shall not include a *kind-param*.
- C417 (R422) A *type-param-value* in a *char-length* shall be a colon, asterisk, or *specification-expr*.
- C418 (R420 R421 R422) A *type-param-value* of * shall be used only
- C419 A function name shall not be declared with an asterisk *type-param-value* unless it is of type CHARACTER and is the name of the result of an external function or the name of a dummy function.
- C420 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, elemental, recursive, or pure.
- C421 (R421) The optional comma in a *length-selector* is permitted only in a *declaration-type-spec* in a *type-declaration-stmt*.
- C422 (R421) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.
- C423 (R420) The length specified for a character statement function or for a statement function dummy argument of type character shall be an initialization expression.
- R423 *char-literal-constant* is [*kind-param* _] ' [*rep-char*] ... '
or [*kind-param* _] " [*rep-char*] ... "
- C424 (R423) The value of *kind-param* shall specify a representation method that exists on the processor.
- R424 *logical-literal-constant* is .TRUE. [_ *kind-param*]
or .FALSE. [_ *kind-param*]
- C425 (R424) The value of *kind-param* shall specify a representation method that exists on the processor.
- R425 *bits-literal-constant* is *binary-constant* [_ *kind-param*]

- or** *octal-constant* [*- kind-param*]
or *hex-constant* [*- kind-param*]
R426 *binary-constant* **is** B ' *digit* [*digit*] ... '
or B " *digit* [*digit*] ... "
- C426 (R426) *digit* shall have one of the values 0 or 1.
- R427 *octal-constant* **is** O ' *digit* [*digit*] ... '
or O " *digit* [*digit*] ... "
- C427 (R427) *digit* shall have one of the values 0 through 7.
- R428 *hex-constant* **is** Z ' *hex-digit* [*hex-digit*] ... '
or Z " *hex-digit* [*hex-digit*] ... "
- R429 *hex-digit* **is** *digit*
or A
or B
or C
or D
or E
or F
- R430 *derived-type-def* **is** *derived-type-stmt*
[*type-param-def-stmt*] ...
[*private-or-sequence*] ...
[*component-part*]
[*type-bound-procedure-part*]
end-type-stmt
- R431 *derived-type-stmt* **is** TYPE [[, *type-attr-spec-list*] ::] *type-name* ■
■ [(*type-param-name-list*)]
- R432 *type-attr-spec* **is** ABSTRACT
or *access-spec*
or BIND (C)
or EXTENDS (*parent-type-name*)
- C428 (R431) A derived type *type-name* shall not be DOUBLEPRECISION or the same as the name of any intrinsic type defined in this part of ISO/IEC 1539.
- C429 (R431) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.
- C430 (R432) A *parent-type-name* shall be the name of a previously defined extensible type (4.5.7).
- C431 (R430) If the type definition contains or inherits (4.5.7.2) a deferred binding (4.5.5), ABSTRACT shall appear.
- C432 (R430) If ABSTRACT appears, the type shall be extensible.
- C433 (R430) If EXTENDS appears, SEQUENCE shall not appear.
- C434 (R430) If EXTENDS appears and the type being defined has a co-array ultimate component, its parent type shall have a co-array ultimate component.
- R433 *private-or-sequence* **is** *private-components-stmt*
or *sequence-stmt*
- C435 (R430) The same *private-or-sequence* shall not appear more than once in a given *derived-type-def*.
- R434 *end-type-stmt* **is** END TYPE [*type-name*]
- C436 (R434) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in the corresponding *derived-type-stmt*.

- R435 *sequence-stmt* is SEQUENCE
- C437 (R439) If SEQUENCE appears, each data component shall be declared to be of an intrinsic type or of a sequence derived type.
- C438 (R430) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.
- R436 *type-param-def-stmt* is INTEGER [*kind-selector*] , *type-param-attr-spec* :: ■
■ *type-param-decl-list*
- R437 *type-param-decl* is *type-param-name* [= *scalar-int-initialization-expr*]
- C439 (R436) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the *type-param-names* in the *derived-type-stmt* of that *derived-type-def*.
- C440 (R436) Each *type-param-name* in the *derived-type-stmt* in a *derived-type-def* shall appear as a *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.
- R438 *type-param-attr-spec* is KIND
or LEN
- R439 *component-part* is [*component-def-stmt*] ...
- R440 *component-def-stmt* is *data-component-def-stmt*
or *proc-component-def-stmt*
- R441 *data-component-def-stmt* is *declaration-type-spec* [[, *component-attr-spec-list*] ::] ■
■ *component-decl-list*
- R442 *component-attr-spec* is *access-spec*
or ALLOCATABLE
or DIMENSION (*component-array-spec*)
or DIMENSION [(*deferred-shape-spec-list*)] ■
■ *lbracket co-array-spec rbracket*
or CONTIGUOUS
or POINTER
- R443 *component-decl* is *component-name* [(*component-array-spec*)] ■
■ [*lbracket co-array-spec rbracket*] ■
■ [* *char-length*] [*component-initialization*]
- R444 *component-array-spec* is *explicit-shape-spec-list*
or *deferred-shape-spec-list*
- C441 (R441) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.
- C442 (R441) If neither the POINTER nor ALLOCATABLE attribute is specified, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.
- C443 (R441) If the POINTER or ALLOCATABLE attribute is specified, the *declaration-type-spec* in the *component-def-stmt* shall be CLASS(*) or shall specify an intrinsic type or any accessible derived type including the type being defined.
- C444 (R441) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.
- C445 (R441) If a *co-array-spec* appears, it shall be a *deferred-co-shape-spec-list* and the component shall have the ALLOCATABLE attribute.
- C446 (R441) If a *co-array-spec* appears, the component shall not be of type IMAGE.TEAM (13.8.2.7), C_PTR, or C_FUNPTR (15.3.3).
- C447 A data component whose type has a co-array ultimate component shall be a nonpointer nonallocatable scalar and shall not be a co-array.
- C448 (R441) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.
- C449 (R444) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a

specification expression that does not contain references to specification functions or any object designators other than named constants or subobjects thereof.

- C450 (R441) A component shall not have both the ALLOCATABLE and the POINTER attribute.
- C451 (R441) If the CONTIGUOUS attribute is specified, the component shall be an array with the POINTER attribute.
- C452 (R443) The * *char-length* option is permitted only if the component is of type character.
- C453 (R440) Each *type-param-value* within a *component-def-stmt* shall either be a colon, be an initialization expression, or be a specification expression that contains neither references to specification functions nor any object designators other than named constants or subobjects thereof.
- R445 *proc-component-def-stmt* **is** PROCEDURE ([*proc-interface*]) , ■
 ■ *proc-component-attr-spec-list* :: *proc-decl-list*
- R446 *proc-component-attr-spec* **is** POINTER
 or PASS [(*arg-name*)]
 or NOPASS
 or *access-spec*
- C454 (R445) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.
- C455 (R445) POINTER shall appear in each *proc-component-attr-spec-list*.
- C456 (R445) If the procedure pointer component has an implicit interface or has no arguments, NOPASS shall be specified.
- C457 (R445) If PASS (*arg-name*) appears, the interface shall have a dummy argument named *arg-name*.
- C458 (R445) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.
- C459 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data object with the same declared type as the type being defined; all of its length type parameters shall be assumed; it shall be polymorphic (4.3.1.3) if and only if the type being defined is extensible (4.5.7). It shall not have the VALUE attribute.
- R447 *component-initialization* **is** = *initialization-expr*
 or => *null-init*
 or => *initial-data-target*
- R448 *initial-data-target* **is** *designator*
- C460 (R441) If *component-initialization* appears, a double-colon separator shall appear before the *component-decl-list*.
- C461 (R441) If *component-initialization* appears, every type parameter and array bound of the component shall be a colon or initialization expression.
- C462 (R441) If => appears in *component-initialization*, POINTER shall appear in the *component-attr-spec-list*. If = appears in *component-initialization*, neither POINTER nor ALLOCATABLE shall appear in the *component-attr-spec-list*.
- C463 (R447) If *initial-data-target* appears, *component-name* shall be data-pointer-initialization compatible with it.
- C464 (R448) The *designator* shall designate a variable that is an initialization target. Every subscript, section subscript, substring starting point, and substring ending point in *designator* shall be an initialization expression.
- R449 *private-components-stmt* **is** PRIVATE
- C465 (R449) A *private-components-stmt* is permitted only if the type definition is within the specification part of a module.
- R450 *type-bound-procedure-part* **is** *contains-stmt*
 [*binding-private-stmt*]
 [*type-bound-proc-binding*] ...

- R451 *binding-private-stmt* **is** PRIVATE
- C466 (R450) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.
- R452 *type-bound-proc-binding* **is** *type-bound-procedure-stmt*
or *type-bound-generic-stmt*
or *final-procedure-stmt*
- R453 *type-bound-procedure-stmt* **is** PROCEDURE [[, *binding-attr-list*] ::] ■
■ *binding-name* [=> *procedure-name*]
or PROCEDURE (*interface-name*) ■
■ , *binding-attr-list* :: *binding-name*
- C467 (R453) If => *procedure-name* appears, the double-colon separator shall appear.
- C468 (R453) The *procedure-name* shall be the name of an accessible module procedure or an external procedure that has an explicit interface.
- R454 *type-bound-generic-stmt* **is** GENERIC ■
■ [, *access-spec*] :: *generic-spec* => *binding-name-list*
- C469 (R454) 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.
- C470 (R454) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the type.
- C471 (R454) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object dummy argument (4.5.4.4).
- C472 (R454) If *generic-spec* is OPERATOR (*defined-operator*), the interface of each binding shall be as specified in 12.4.3.4.2.
- C473 (R454) If *generic-spec* is ASSIGNMENT (=), the interface of each binding shall be as specified in 12.4.3.4.3.
- C474 (R454) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in 9.5.4.7. The type of the *dtv* argument shall be *type-name*.
- R455 *binding-attr* **is** PASS [(*arg-name*)]
or NOPASS
or NON_OVERRIDABLE
or DEFERRED
or *access-spec*
- C475 (R455) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.
- C476 (R453) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.
- C477 (R453) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument named *arg-name*.
- C478 (R455) PASS and NOPASS shall not both appear in the same *binding-attr-list*.
- C479 (R455) NON_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-attr-list*.
- C480 (R455) DEFERRED shall appear if and only if *interface-name* appears.
- C481 (R453) An overriding binding (4.5.7.3) shall have the DEFERRED attribute only if the binding it overrides is deferred.
- C482 (R453) A binding shall not override an inherited binding (4.5.7.2) that has the NON_OVERRIDABLE attribute.
- R456 *final-procedure-stmt* **is** FINAL [::] *final-subroutine-name-list*
- C483 (R456) A *final-subroutine-name* shall be the name of a module procedure with exactly one

- dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocatable, nonpolymorphic variable of the derived type being defined. All length type parameters of the dummy argument shall be assumed. The dummy argument shall not have INTENT(OUT).
- C484 (R456) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.
- C485 (R456) A final subroutine shall not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of that type.
- R457 *derived-type-spec* **is** *type-name* [(*type-param-spec-list*)]
- R458 *type-param-spec* **is** [*keyword* =] *type-param-value*
- C486 (R457) *type-name* shall be the name of an accessible derived type.
- C487 (R457) *type-param-spec-list* shall appear only if the type is parameterized.
- C488 (R457) 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.
- C489 (R458) The *keyword*= may be omitted from a *type-param-spec* only if the *keyword*= has been omitted from each preceding *type-param-spec* in the *type-param-spec-list*.
- C490 (R458) Each *keyword* shall be the name of a parameter of the type.
- C491 (R458) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the declaration of a dummy argument or associate name or in the allocation of a dummy argument.
- R459 *structure-constructor* **is** *derived-type-spec* ([*component-spec-list*])
- R460 *component-spec* **is** [*keyword* =] *component-data-source*
- R461 *component-data-source* **is** *expr*
 or *data-target*
 or *proc-target*
- C492 (R459) The *derived-type-spec* shall not specify an abstract type (4.5.7).
- C493 (R459) At most one *component-spec* shall be provided for a component.
- C494 (R459) 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.
- C495 (R459) A *component-spec* shall be provided for a nonallocatable component unless it has default initialization or is inheritance associated with a subcomponent of another component for which a *component-spec* is provided.
- C496 (R460) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been omitted from each preceding *component-spec* in the constructor.
- C497 (R460) Each *keyword* shall be the name of a component of the type.
- C498 (R459) The type name and all components of the type for which a *component-spec* appears shall be accessible in the scoping unit containing the structure constructor.
- C499 (R459) If *derived-type-spec* is a type name that is the same as a generic name, the *component-spec-list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a generic reference to that name (12.5.5.2).
- C4100 (R461) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall correspond to a procedure pointer component.
- C4101 (R461) A *data-target* shall have the same rank as its corresponding component.
- R462 *enum-def* **is** *enum-def-stmt*
 enumerator-def-stmt
 [*enumerator-def-stmt*] ...
 end-enum-stmt
- R463 *enum-def-stmt* **is** ENUM, BIND(C)
- R464 *enumerator-def-stmt* **is** ENUMERATOR [::] *enumerator-list*

- R465 *enumerator* **is** *named-constant* [= *scalar-int-initialization-expr*]
- R466 *end-enum-stmt* **is** END ENUM
- C4102 (R464) If = appears in an *enumerator*, a double-colon separator shall appear before the *enumerator-list*.
- R467 *array-constructor* **is** (/ *ac-spec* /)
or *lbracket ac-spec rbracket*
- R468 *ac-spec* **is** *type-spec* ::
or [*type-spec* ::] *ac-value-list*
- R469 *lbracket* **is** [
R470 *rbracket* **is**]
R471 *ac-value* **is** *expr*
or *ac-implied-do*
- R472 *ac-implied-do* **is** (*ac-value-list* , *ac-implied-do-control*)
- R473 *ac-implied-do-control* **is** *ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■
■ [, *scalar-int-expr*]
- R474 *ac-do-variable* **is** *do-variable*
- C4103 (R468) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameters.
- C4104 (R468) If *type-spec* specifies an intrinsic type, each *ac-value* expression in the *array-constructor* shall be of an intrinsic type that is in type conformance with a variable of type *type-spec* as specified in Table 7.12.
- C4105 (R468) If *type-spec* specifies a derived type, all *ac-value* expressions in the *array-constructor* shall be of that derived type and shall have the same kind type parameter values as specified by *type-spec*.
- C4106 (R472) The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear as the *ac-do-variable* of the containing *ac-implied-do*.

Clause 5:

- R501 *type-declaration-stmt* **is** *declaration-type-spec* [[, *attr-spec*] ... ::] *entity-decl-list*
- R502 *attr-spec* **is** *access-spec*
or ALLOCATABLE
or ASYNCHRONOUS
or CONTIGUOUS
or dimension-spec
or EXTERNAL
or INTENT (*intent-spec*)
or INTRINSIC
or *language-binding-spec*
or OPTIONAL
or PARAMETER
or POINTER
or PROTECTED
or SAVE
or TARGET
or VALUE
or VOLATILE
- C501 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.
- C502 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall

consist of a single *entity-decl*.

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

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

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

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

C506 (R503) An *initialization* shall not appear if *object-name* is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, an allocatable variable, an external function, an intrinsic function, or an automatic object.

C507 (R503) An *initialization* shall appear if the entity is a named constant (5.3.12).

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

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

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

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

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

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

C511 (R503) If *initial-data-target* appears, *object-name* shall be data-pointer-initialization compatible with it (4.5.4.5).

C512 (R506) The *function-reference* shall be a reference to the NULL intrinsic function with no arguments.

C513 An automatic object shall not be a local variable of a main program, module, or submodule.

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

C515 An *array-spec* for a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.

C516 The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*.

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

C517 (R507) An *access-spec* shall appear only in the *specification-part* of a module.

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

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

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

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

C521 Each variable of a common block with the BIND attribute shall be interoperable.

C522 (R508) The *scalar-char-initialization-expr* shall be of default character kind.

C523 An entity with the CONTIGUOUS attribute shall be an array pointer or an assumed-shape array.

R509 *dimension-spec* **is** DIMENSION (*array-spec*)
 or DIMENSION [(*array-spec*)] *lbracket co-array-spec rbracket*

C524 (R501) A co-array with the ALLOCATABLE attribute shall be specified with a *co-array-spec*

that is a *deferred-co-shape-spec-list*.

- C525 A co-array shall be a component or a variable that is not a function result.
- C526 A co-array shall not be of type IMAGE_TEAM (13.8.2.7), C_PTR, or C_FUNPTR (15.3.3).
- C527 An entity whose type has a co-array ultimate component shall be a nonpointer nonallocatable scalar, shall not be a co-array, and shall not be a function result.
- C528 The SAVE attribute shall be specified for a co-array unless it is a dummy argument, declared in the main program, or allocatable.
- R510 *array-spec* **is** *explicit-shape-spec-list*
 or *assumed-shape-spec-list*
 or *deferred-shape-spec-list*
 or *assumed-size-spec*
 or *implied-shape-spec-list*
- R511 *co-array-spec* **is** *deferred-co-shape-spec-list*
 or *explicit-co-shape-spec*
- C529 The sum of the rank and co-rank of an entity shall not exceed fifteen.
- R512 *explicit-shape-spec* **is** [*lower-bound* :] *upper-bound*
- R513 *lower-bound* **is** *specification-expr*
- R514 *upper-bound* **is** *specification-expr*
- C530 (R512) An *explicit-shape-spec* whose bounds are not initialization expressions shall appear only in a subprogram or interface body.
- R515 *assumed-shape-spec* **is** [*lower-bound*] :
- R516 *deferred-shape-spec* **is** :
- C531 An array with the POINTER or ALLOCATABLE attribute shall have an *array-spec* that is a *deferred-shape-spec-list*.
- R517 *assumed-size-spec* **is** [*explicit-shape-spec* ,]... [*lower-bound* :] *
- C532 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy data argument.
- C533 An assumed-size array with INTENT (OUT) shall not be polymorphic, of a finalizable type, of a type with an ultimate allocatable component, or of a type for which default initialization is specified.
- R518 *implied-shape-spec* **is** [*lower-bound* :] *
- C534 An implied-shape array shall be a named constant.
- R519 *deferred-co-shape-spec* **is** :
- C535 A co-array with the ALLOCATABLE attribute shall have a *co-array-spec* that is a *deferred-co-shape-spec-list*.
- R520 *explicit-co-shape-spec* **is** [[*lower-co-bound* :] *upper-co-bound* ,]... ■
 ■ [*lower-co-bound* :] *
- C536 A co-array that does not have the ALLOCATABLE attribute shall have a *co-array-spec* that is an *explicit-co-shape-spec*.
- R521 *lower-co-bound* **is** *specification-expr*
- R522 *upper-co-bound* **is** *specification-expr*
- C537 (R520) A *lower-co-bound* or *upper-co-bound* that is not an initialization expression shall appear only in a subprogram or interface body.
- C538 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.
- C539 In an external subprogram, the EXTERNAL attribute shall not be specified for a procedure defined by the subprogram.
- R523 *intent-spec* **is** IN
 or OUT

- R526 *allocatable-stmt* is ALLOCATABLE [::] *allocatable-decl-list*
- R527 *allocatable-decl* is *object-name* [(*array-spec*)] ■
 ■ [*lbracket co-array-spec rbracket*]
- R528 *asynchronous-stmt* is ASYNCHRONOUS [::] *object-name-list*
- R529 *bind-stmt* is *language-binding-spec* [::] *bind-entity-list*
- R530 *bind-entity* is *entity-name*
 or / *common-block-name* /
- C563 (R529) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of a single *bind-entity*.
- R531 *contiguous-stmt* is CONTIGUOUS [::] *object-name-list*
- R532 *data-stmt* is DATA *data-stmt-set* [[,] *data-stmt-set*] ...
- R533 *data-stmt-set* is *data-stmt-object-list* / *data-stmt-value-list* /
- R534 *data-stmt-object* is *variable*
 or *data-implied-do*
- R535 *data-implied-do* is (*data-i-do-object-list* , *data-i-do-variable* = ■
 ■ *scalar-int-initialization-expr* , ■
 ■ *scalar-int-initialization-expr* ■
 ■ [, *scalar-int-initialization-expr*])
- R536 *data-i-do-object* is *array-element*
 or *scalar-structure-component*
 or *data-implied-do*
- R537 *data-i-do-variable* is *do-variable*
- C564 A *data-stmt-object* or *data-i-do-object* shall not be a co-indexed variable.
- C565 (R534) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring starting point, and substring ending point shall be an initialization expression.
- C566 (R534) A variable whose designator appears as a *data-stmt-object* or a *data-i-do-object* shall not be a dummy argument, made accessible by use association or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, a function name, a function result name, an automatic object, or an allocatable variable.
- C567 (R534) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object designator in which a pointer appears other than as the entire rightmost *part-ref*.
- C568 (R536) The *array-element* shall be a variable.
- C569 (R536) The *scalar-structure-component* shall be a variable.
- C570 (R536) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.
- C571 (R536) In an *array-element* or *scalar-structure-component* that is a *data-i-do-object*, any subscript shall be an initialization expression, and any primary within that subscript that is a *data-i-do-variable* shall be a DO variable of this *data-implied-do* or of a containing *data-implied-do*.
- R538 *data-stmt-value* is [*data-stmt-repeat* *] *data-stmt-constant*
- R539 *data-stmt-repeat* is *scalar-int-constant*
 or *scalar-int-constant-subobject*
- C572 (R539) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named constant, it shall have been declared previously in the scoping unit or made accessible by use association or host association.
- R540 *data-stmt-constant* is *scalar-constant*
 or *scalar-constant-subobject*
 or *signed-int-literal-constant*

- or *signed-real-literal-constant*
- or *null-init*
- or *initial-data-target*
- or *structure-constructor*
- C573 (R540) If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type shall have been declared previously in the scoping unit or made accessible by use or host association.
- C574 (R540) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.
- R541 *int-constant-subobject* is *constant-subobject*
- C575 (R541) *int-constant-subobject* shall be of type integer.
- R542 *constant-subobject* is *designator*
- C576 (R542) *constant-subobject* shall be a subobject of a constant.
- C577 (R542) Any subscript, substring starting point, or substring ending point shall be an initialization expression.
- R543 *dimension-stmt* is DIMENSION [::] *dimension-decl-list*
- R544 *dimension-decl* is *array-name* (*array-spec*)
or *co-name* [(*array-spec*)] *lbracket co-array-spec rbracket*
- R545 *intent-stmt* is INTENT (*intent-spec*) [::] *dummy-arg-name-list*
- R546 *optional-stmt* is OPTIONAL [::] *dummy-arg-name-list*
- R547 *parameter-stmt* is PARAMETER (*named-constant-def-list*)
- R548 *named-constant-def* is *named-constant* = *initialization-expr*
- R549 *pointer-stmt* is POINTER [::] *pointer-decl-list*
- R550 *pointer-decl* is *object-name* [(*deferred-shape-spec-list*)]
or *proc-entity-name*
- R551 *protected-stmt* is PROTECTED [::] *entity-name-list*
- R552 *save-stmt* is SAVE [[::] *saved-entity-list*]
- R553 *saved-entity* is *object-name*
or *proc-pointer-name*
or / *common-block-name* /
- R554 *proc-pointer-name* is *name*
- C578 (R552) 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.
- R555 *target-stmt* is TARGET [::] *target-decl-list*
- R556 *target-decl* is *object-name* [(*array-spec*)] ■
■ [*lbracket co-array-spec rbracket*]
- R557 *value-stmt* is VALUE [::] *dummy-arg-name-list*
- R558 *volatile-stmt* is VOLATILE [::] *object-name-list*
- R559 *implicit-stmt* is IMPLICIT *implicit-spec-list*
or IMPLICIT NONE
- R560 *implicit-spec* is *declaration-type-spec* (*letter-spec-list*)
- R561 *letter-spec* is *letter* [– *letter*]
- C579 (R559) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER statements that appear in the scoping unit and there shall be no other IMPLICIT statements in the scoping unit.
- C580 (R561) If the minus and second *letter* appear, the second letter shall follow the first letter alphabetically.
- R562 *namelist-stmt* is NAMELIST ■

■ / *namelist-group-name* / *namelist-group-object-list* ■
 ■ [[,] / *namelist-group-name* / ■
 ■ *namelist-group-object-list*] ...

- C581 (R562) The *namelist-group-name* shall not be a name accessed by use association.
- R563 *namelist-group-object* **is** *variable-name*
- C582 (R563) A *namelist-group-object* shall not be an assumed-size array.
- C583 (R562) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.
- R564 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*
- R565 *equivalence-set* **is** (*equivalence-object* , *equivalence-object-list*)
- R566 *equivalence-object* **is** *variable-name*
or *array-element*
or *substring*
- C584 (R566) An *equivalence-object* shall not be a designator with a base object that is a dummy argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ultimate component, an object of a nonsequence derived type, an object of a derived type that has a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.
- C585 (R566) An *equivalence-object* shall not be a designator that has more than one *part-ref*.
- C586 (R566) An *equivalence-object* shall not be a co-array or a subobject thereof.
- C587 (R566) An *equivalence-object* shall not have the TARGET attribute.
- C588 (R566) Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.12).
- C589 (R565) If an *equivalence-object* is of type default integer, default real, double precision real, default complex, default logical, default bits, or numeric sequence type, all of the objects in the equivalence set shall be of these types.
- C590 (R565) If an *equivalence-object* is of type default character or character sequence type, all of the objects in the equivalence set shall be of these types.
- C591 (R565) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or character sequence type, all of the objects in the equivalence set shall be of the same type with the same type parameter values.
- C592 (R565) If an *equivalence-object* is of an intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character, all of the objects in the equivalence set shall be of the same type with the same kind type parameter value.
- C593 (R566) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equivalence set shall have the PROTECTED attribute.
- C594 (R566) The name of an *equivalence-object* shall not be a name made accessible by use association.
- C595 (R566) A *substring* shall not have length zero.
- R567 *common-stmt* **is** COMMON ■
 ■ [/ [*common-block-name*] /] *common-block-object-list* ■
 ■ [[,] / [*common-block-name*] / ■
 ■ *common-block-object-list*] ...
- R568 *common-block-object* **is** *variable-name* [(*array-spec*)]
or *proc-pointer-name*
- C596 (R568) An *array-spec* in a *common-block-object* shall be an *explicit-shape-spec-list*.
- C597 (R568) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all *common-block-object-lists* within a scoping unit.
- C598 (R568) A *common-block-object* shall not be a dummy argument, an allocatable variable, a

derived-type object with an ultimate component that is allocatable, an automatic object, a function name, an entry name, a variable with the BIND attribute, a co-array, or a result name.

C599 (R568) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.2.3) or a type with the BIND attribute and it shall have no default initialization.

C5100 (R568) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use association.

Clause 6:

R601 *variable* **is** *designator*
or *expr*

C601 (R601) *designator* shall not be a constant or a subobject of a constant.

C602 (R601) *expr* shall be a reference to a function that has a pointer result.

R602 *variable-name* **is** *name*

C603 (R602) *variable-name* shall be the name of a variable.

R603 *designator* **is** *object-name*
or *array-element*
or *array-section*
or *complex-part-designator*
or *structure-component*
or *substring*

R604 *logical-variable* **is** *variable*

C604 (R604) *logical-variable* shall be of type logical.

R605 *default-logical-variable* **is** *variable*

C605 (R605) *default-logical-variable* shall be of type default logical.

R606 *char-variable* **is** *variable*

C606 (R606) *char-variable* shall be of type character.

R607 *default-char-variable* **is** *variable*

C607 (R607) *default-char-variable* shall be of type default character.

R608 *int-variable* **is** *variable*

C608 (R608) *int-variable* shall be of type integer.

R609 *substring* **is** *parent-string* (*substring-range*)

R610 *parent-string* **is** *scalar-variable-name*
or *array-element*
or *scalar-structure-component*
or *scalar-constant*

R611 *substring-range* **is** [*scalar-int-expr*] : [*scalar-int-expr*]

C609 (R610) *parent-string* shall be of type character.

R612 *data-ref* **is** *part-ref* [% *part-ref*] ...

R613 *part-ref* **is** *part-name* [(*section-subscript-list*)] [*image-selector*]

C610 (R612) Each *part-name* except the rightmost shall be of derived type.

C611 (R612) Each *part-name* except the leftmost shall be the name of a component of the declared type of the preceding *part-name*.

C612 (R612) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.

C613 (R612) The leftmost *part-name* shall be the name of a data object.

C614 (R613) If a *section-subscript-list* appears, the number of *section-subscripts* shall equal the rank of *part-name*.

C615 (R613) If *image-selector* appears, the number of *co-subscripts* shall be equal to the co-rank of

- part-name*.
- C616 (R613) If *image-selector* appears and *part-name* is an array, *section-subscript-list* shall appear.
- C617 (R612) If *image-selector* appears, *data-ref* shall not be, or have a direct component, of type IMAGE_TEAM (13.8.2.7), C_PTR, or C_FUNPTR (15.3.3).
- C618 (R612) A *data-ref* that is a co-indexed object shall not have a pointer ultimate component.
- C619 (R612) 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.
- R614 *structure-component* **is** *data-ref*
- C620 (R614) There shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form *part-name*.
- R615 *complex-part-designator* **is** *designator* % RE
 or *designator* % IM
- C621 (R615) The *designator* shall be of complex type.
- R616 *type-param-inquiry* **is** *designator* % *type-param-name*
- C622 (R616) The *type-param-name* shall be the name of a type parameter of the declared type of the object designated by the *designator*.
- R617 *array-element* **is** *data-ref*
- C623 (R617) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.
- R618 *array-section* **is** *data-ref* [(*substring-range*)]
 or *complex-part-designator*
- C624 (R618) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a *section-subscript-list* with nonzero rank, another *part-ref* shall have nonzero rank, or the *complex-part-designator* shall be an array.
- C625 (R618) If a *substring-range* appears, the rightmost *part-name* shall be of type character.
- R619 *subscript* **is** *scalar-int-expr*
- R620 *section-subscript* **is** *subscript*
 or *subscript-triplet*
 or *vector-subscript*
- R621 *subscript-triplet* **is** [*subscript*] : [*subscript*] [: *stride*]
- R622 *stride* **is** *scalar-int-expr*
- R623 *vector-subscript* **is** *int-expr*
- C626 (R623) A *vector-subscript* shall be an integer array expression of rank one.
- C627 (R621) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension of an assumed-size array.
- R624 *image-selector* **is** *lbracket co-subscript-list rbracket*
- R625 *co-subscript* **is** *scalar-int-expr*
- R626 *allocate-stmt* **is** ALLOCATE ([*type-spec* ::] *allocation-list* ■
 ■ [, *alloc-opt-list*])
- R627 *alloc-opt* **is** ERRMSG = *errmsg-variable*
 or MOLD = *source-expr*
 or SOURCE = *source-expr*
 or STAT = *stat-variable*
- R628 *stat-variable* **is** *scalar-int-variable*
- R629 *errmsg-variable* **is** *scalar-default-char-variable*
- R630 *source-expr* **is** *expr*
- R631 *allocation* **is** *allocate-object* [(*allocate-shape-spec-list*)] ■
 ■ [*lbracket allocate-co-array-spec rbracket*]

- R632 *allocate-object* **is** *variable-name*
or *structure-component*
- R633 *allocate-shape-spec* **is** [*lower-bound-expr* :] *upper-bound-expr*
- R634 *lower-bound-expr* **is** *scalar-int-expr*
- R635 *upper-bound-expr* **is** *scalar-int-expr*
- R636 *allocate-co-array-spec* **is** [*allocate-co-shape-spec-list* ,] [*lower-bound-expr* :] *
- R637 *allocate-co-shape-spec* **is** [*lower-bound-expr* :] *upper-bound-expr*
- C628 (R632) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.
- C629 (R626) If any *allocate-object* has a deferred type parameter, is unlimited polymorphic, or is of abstract type, either *type-spec* or *source-expr* shall appear.
- C630 (R626) If a *type-spec* appears, it shall specify a type with which each *allocate-object* is type compatible.
- C631 (R626) A *type-param-value* in a *type-spec* shall be an asterisk if and only if each *allocate-object* is a dummy argument for which the corresponding type parameter is assumed.
- C632 (R626) If a *type-spec* appears, the kind type parameter values of each *allocate-object* shall be the same as the corresponding type parameter values of the *type-spec*.
- C633 (R631) If *allocate-object* is an array either *allocate-shape-spec-list* shall appear or *source-expr* shall appear and have the same rank as *allocate-object*. If *allocate-object* is scalar, *allocate-shape-spec-list* shall not appear.
- C634 (R631) An *allocate-co-array-spec* shall appear if and only if the *allocate-object* is a co-array.
- C635 (R631) The number of *allocate-shape-specs* in an *allocate-shape-spec-list* shall be the same as the rank of the *allocate-object*. The number of *allocate-co-shape-specs* in an *allocate-co-array-spec* shall be one less than the co-rank of the *allocate-object*.
- C636 (R627) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.
- C637 (R626) At most one of *source-expr* and *type-spec* shall appear.
- C638 (R626) Each *allocate-object* shall be type compatible (4.3.1.3) with *source-expr*. If SOURCE= appears, *source-expr* shall be a scalar or have the same rank as each *allocate-object*.
- C639 (R626) Corresponding kind type parameters of *allocate-object* and *source-expr* shall have the same values.
- C640 (R626) *type-spec* shall not specify a type that has a co-array ultimate component.
- C641 (R630) The declared type of *source-expr* shall not have a co-array ultimate component.
- C642 (R632) An *allocate-object* shall not be a co-indexed object.
- R638 *nullify-stmt* **is** NULLIFY (*pointer-object-list*)
- R639 *pointer-object* **is** *variable-name*
or *structure-component*
or *proc-pointer-name*
- C643 (R639) Each *pointer-object* shall have the POINTER attribute.
- R640 *deallocate-stmt* **is** DEALLOCATE (*allocate-object-list* [, *dealloc-opt-list*])
- C644 (R640) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.
- R641 *dealloc-opt* **is** STAT = *stat-variable*
or ERRMSG = *errmsg-variable*
- C645 (R641) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

Clause 7:

- R701 *primary* **is** *constant*
or *designator*
or *array-constructor*
or *structure-constructor*

		or <i>function-reference</i>
		or <i>type-param-inquiry</i>
		or <i>type-param-name</i>
		or (<i>expr</i>)
C701	(R701)	The <i>type-param-name</i> shall be the name of a type parameter.
C702	(R701)	The <i>designator</i> shall not be a whole assumed-size array.
R702	<i>level-1-expr</i>	is [<i>defined-unary-op</i>] <i>primary</i>
R703	<i>defined-unary-op</i>	is . <i>letter</i> [<i>letter</i>]
C703	(R703)	A <i>defined-unary-op</i> shall not contain more than 63 letters and shall not be the same as any <i>intrinsic-operator</i> or <i>logical-literal-constant</i> .
R704	<i>mult-operand</i>	is <i>level-1-expr</i> [<i>power-op mult-operand</i>]
R705	<i>add-operand</i>	is [<i>add-operand mult-op</i>] <i>mult-operand</i>
R706	<i>level-2-expr</i>	is [[<i>level-2-expr</i>] <i>add-op</i>] <i>add-operand</i>
R707	<i>power-op</i>	is **
R708	<i>mult-op</i>	is *
		or /
R709	<i>add-op</i>	is +
		or -
R710	<i>level-3-expr</i>	is [<i>level-3-expr concat-op</i>] <i>level-2-expr</i>
R711	<i>concat-op</i>	is //
R712	<i>level-4-expr</i>	is [<i>level-3-expr rel-op</i>] <i>level-3-expr</i>
R713	<i>rel-op</i>	is .EQ.
		or .NE.
		or .LT.
		or .LE.
		or .GT.
		or .GE.
		or ==
		or /=
		or <
		or <=
		or >
		or >=
R714	<i>and-operand</i>	is [<i>not-op</i>] <i>level-4-expr</i>
R715	<i>or-operand</i>	is [<i>or-operand and-op</i>] <i>and-operand</i>
R716	<i>equiv-operand</i>	is [<i>equiv-operand or-op</i>] <i>or-operand</i>
R717	<i>level-5-expr</i>	is [<i>level-5-expr equiv-op</i>] <i>equiv-operand</i>
R718	<i>not-op</i>	is .NOT.
R719	<i>and-op</i>	is .AND.
R720	<i>or-op</i>	is .OR.
R721	<i>equiv-op</i>	is .EQV.
		or .NEQV.
		or .XOR.
R722	<i>expr</i>	is [<i>expr defined-binary-op</i>] <i>level-5-expr</i>
R723	<i>defined-binary-op</i>	is . <i>letter</i> [<i>letter</i>]
C704	(R723)	A <i>defined-binary-op</i> shall not contain more than 63 letters and shall not be the same as any <i>intrinsic-operator</i> or <i>logical-literal-constant</i> .

- R724 *logical-expr* **is** *expr*
 C705 (R724) *logical-expr* shall be of type logical.
- R725 *char-expr* **is** *expr*
 C706 (R725) *char-expr* shall be of type character.
- R726 *default-char-expr* **is** *expr*
 C707 (R726) *default-char-expr* shall be of type default character.
- R727 *int-expr* **is** *expr*
 C708 (R727) *int-expr* shall be of type integer.
- R728 *numeric-expr* **is** *expr*
 C709 (R728) *numeric-expr* shall be of type integer, real, or complex.
- C710 The kind type parameter of the result of a bits concatenation operation expression shall be a bits kind type parameter value supported by the processor.
- R729 *specification-expr* **is** *scalar-int-expr*
 C711 (R729) The *scalar-int-expr* shall be a restricted expression.
- R730 *initialization-expr* **is** *expr*
 C712 (R730) *initialization-expr* shall be an initialization expression.
- R731 *char-initialization-expr* **is** *char-expr*
 C713 (R731) *char-initialization-expr* shall be an initialization expression.
- R732 *int-initialization-expr* **is** *int-expr*
 C714 (R732) *int-initialization-expr* shall be an initialization expression.
- R733 *logical-initialization-expr* **is** *logical-expr*
 C715 (R733) *logical-initialization-expr* shall be an initialization expression.
- R734 *assignment-stmt* **is** *variable = expr*
 C716 (R734) The *variable* shall not be a whole assumed-size array.
- R735 *pointer-assignment-stmt* **is** *data-pointer-object* [(*bounds-spec-list*)] => *data-target*
or *data-pointer-object* (*bounds-remapping-list*) => *data-target*
or *proc-pointer-object* => *proc-target*
- R736 *data-pointer-object* **is** *variable-name*
or *scalar-variable* % *data-pointer-component-name*
- C717 (R735) If *data-target* is not unlimited polymorphic, *data-pointer-object* shall be type compatible (4.3.1.3) with it and the corresponding kind type parameters shall be equal.
- C718 (R735) If *data-target* is unlimited polymorphic, *data-pointer-object* shall be unlimited polymorphic, of a sequence derived type, or of a type with the BIND attribute.
- C719 (R735) If *bounds-spec-list* is specified, the number of *bounds-specs* shall equal the rank of *data-pointer-object*.
- C720 (R735) If *bounds-remapping-list* is specified, the number of *bounds-remappings* shall equal the rank of *data-pointer-object*.
- C721 (R735) If *bounds-remapping-list* is not specified, the ranks of *data-pointer-object* and *data-target* shall be the same.
- C722 (R736) A *variable-name* shall have the POINTER attribute.
- C723 (R736) A *scalar-variable* shall be a *data-ref*.
- C724 (R736) A *data-pointer-component-name* shall be the name of a component of *scalar-variable* that is a data pointer.
- C725 (R736) A *data-pointer-object* shall not be a co-indexed object.
- R737 *bounds-spec* **is** *lower-bound-expr* :
- R738 *bounds-remapping* **is** *lower-bound-expr* : *upper-bound-expr*
- R739 *data-target* **is** *variable*

- or** *expr*
- C726 (R739) A *variable* shall have either the TARGET or POINTER attribute, and shall not be an array section with a vector subscript.
- C727 (R739) A *data-target* shall not be a co-indexed object.
- C728 (R739) An *expr* shall be a reference to a function whose result is a data pointer.
- R740 *proc-pointer-object* **is** *proc-pointer-name*
or *proc-component-ref*
- R741 *proc-component-ref* **is** *scalar-variable* % *procedure-component-name*
- C729 (R741) The *scalar-variable* shall be a *data-ref*.
- C730 (R741) The *procedure-component-name* shall be the name of a procedure pointer component of the declared type of *scalar-variable*.
- R742 *proc-target* **is** *expr*
or *procedure-name*
or *proc-component-ref*
- C731 (R742) An *expr* shall be a reference to a function whose result is a procedure pointer.
- C732 (R742) A *procedure-name* shall be the name of an external, internal, module, or dummy procedure, a procedure pointer, or a specific intrinsic function listed in 13.6 and not marked with a bullet (●).
- C733 (R742) The *proc-target* shall not be a nonintrinsic elemental procedure.
- R743 *where-stmt* **is** WHERE (*mask-expr*) *where-assignment-stmt*
- R744 *where-construct* **is** *where-construct-stmt*
[*where-body-construct*] ...
[*masked-elsewhere-stmt*
[*where-body-construct*] ...] ...
[*elsewhere-stmt*
[*where-body-construct*] ...]
end-where-stmt
- R745 *where-construct-stmt* **is** [*where-construct-name*:] WHERE (*mask-expr*)
- R746 *where-body-construct* **is** *where-assignment-stmt*
or *where-stmt*
or *where-construct*
- R747 *where-assignment-stmt* **is** *assignment-stmt*
- R748 *mask-expr* **is** *logical-expr*
- R749 *masked-elsewhere-stmt* **is** ELSEWHERE (*mask-expr*) [*where-construct-name*]
- R750 *elsewhere-stmt* **is** ELSEWHERE [*where-construct-name*]
- R751 *end-where-stmt* **is** END WHERE [*where-construct-name*]
- C734 (R747) A *where-assignment-stmt* that is a defined assignment shall be elemental.
- C735 (R744) 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*.
- C736 (R746) A statement that is part of a *where-body-construct* shall not be a branch target statement.
- R752 *forall-construct* **is** *forall-construct-stmt*
[*forall-body-construct*] ...
end-forall-stmt
- R753 *forall-construct-stmt* **is** [*forall-construct-name* :] FORALL *forall-header*

- R754 *forall-header* is ([*type-spec* ::] *forall-triplet-spec-list* [, *scalar-mask-expr*])
- R755 *forall-triplet-spec* is *index-name* = *subscript* : *subscript* [: *stride*]
- R756 *forall-body-construct* is *forall-assignment-stmt*
or *where-stmt*
or *where-construct*
or *forall-construct*
or *forall-stmt*
- R757 *forall-assignment-stmt* is *assignment-stmt*
or *pointer-assignment-stmt*
- R758 *end-forall-stmt* is END FORALL [*forall-construct-name*]
- C737 (R758) If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have the same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-construct-stmt* shall have the same *forall-construct-name*.
- C738 (R754) *type-spec* shall specify type integer.
- C739 (R754) The *scalar-mask-expr* shall be scalar and of type logical.
- C740 (R754) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined operation, shall be a pure procedure (12.7).
- C741 (R755) The *index-name* shall be a named scalar variable of type integer.
- C742 (R755) A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.
- C743 (R756) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.
- C744 (R756) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined operation, assignment, or finalization, shall be a pure procedure.
- C745 (R756) A *forall-body-construct* shall not be a branch target.
- C746 (R757) The *variable* in an *assignment-stmt* that is a *forall-assignment-stmt* shall be a *designator*.
- R759 *forall-stmt* is FORALL *forall-header forall-assignment-stmt*

Clause 8:

- R801 *block* is [*execution-part-construct*] ...
- R802 *associate-construct* is *associate-stmt*
block
end-associate-stmt
- R803 *associate-stmt* is [*associate-construct-name* :] ASSOCIATE ■
■ (*association-list*)
- R804 *association* is *associate-name* => *selector*
- R805 *selector* is *expr*
or *variable*
- C801 (R804) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.6.7).
- C802 (R804) An *associate-name* shall not be the same as another *associate-name* in the same *associate-stmt*.
- C803 (R805) *expr* shall not be a reference to a function that has a pointer result.
- R806 *end-associate-stmt* is END ASSOCIATE [*associate-construct-name*]
- C804 (R806) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*, the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the corresponding *end-associate-stmt* shall not specify an *associate-construct-name*.

- R807 *block-construct* **is** *block-stmt*
 [*specification-part*]
 block
 end-block-stmt
- R808 *block-stmt* **is** [*block-construct-name* :] BLOCK
- R809 *end-block-stmt* **is** END BLOCK [*block-construct-name*]
- C805 (R807) The *specification-part* of a BLOCK construct shall not contain a COMMON, EQUIVALENCE, IMPLICIT, INTENT, NAMELIST, OPTIONAL, or USE statement.
- C806 (R807) A SAVE statement in a BLOCK construct shall contain a *saved-entity-list* that does not specify a *common-block-name*.
- C807 (R807) If the *block-stmt* of a *block-construct* specifies a *block-construct-name*, the corresponding *end-block-stmt* shall specify the same *block-construct-name*. If the *block-stmt* does not specify a *block-construct-name*, the corresponding *end-block-stmt* shall not specify a *block-construct-name*.
- R810 *case-construct* **is** *select-case-stmt*
 [*case-stmt*
 block] ...
 end-select-stmt
- R811 *select-case-stmt* **is** [*case-construct-name* :] SELECT CASE (*case-expr*)
- R812 *case-stmt* **is** CASE *case-selector* [*case-construct-name*]
- R813 *end-select-stmt* **is** END SELECT [*case-construct-name*]
- C808 (R810) 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*.
- R814 *case-expr* **is** *scalar-int-expr*
 or *scalar-char-expr*
 or *scalar-logical-expr*
- R815 *case-selector* **is** (*case-value-range-list*)
 or DEFAULT
- C809 (R810) No more than one of the selectors of one of the CASE statements shall be DEFAULT.
- R816 *case-value-range* **is** *case-value*
 or *case-value* :
 or : *case-value*
 or *case-value* : *case-value*
- R817 *case-value* **is** *scalar-int-initialization-expr*
 or *scalar-char-initialization-expr*
 or *scalar-logical-initialization-expr*
- C810 (R810) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For character type, the kind type parameters shall be the same; character length differences are allowed.
- C811 (R810) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.
- C812 (R810) For a given *case-construct*, the *case-value-ranges* shall not overlap; that is, there shall be no possible value of the *case-expr* that matches more than one *case-value-range*.
- R818 *critical-construct* **is** *critical-stmt*
 block
 end-critical-stmt

- R819 *critical-stmt* **is** [*critical-construct-name* :] CRITICAL
- R820 *end-critical-stmt* **is** END CRITICAL [*critical-construct-name*]
- C813 (R818) If the *critical-stmt* of a *critical-construct* specifies a *critical-construct-name*, the corresponding *end-critical-stmt* shall specify the same *critical-construct-name*. If the *critical-stmt* of a *critical-construct* does not specify a *critical-construct-name*, the corresponding *end-critical-stmt* shall not specify a *critical-construct-name*.
- C814 (R818) The *block* of a *critical-construct* shall not contain an image control statement.
- R821 *do-construct* **is** *block-do-construct*
or *nonblock-do-construct*
- R822 *block-do-construct* **is** *do-stmt*
do-block
end-do
- R823 *do-stmt* **is** *label-do-stmt*
or *nonlabel-do-stmt*
- R824 *label-do-stmt* **is** [*do-construct-name* :] DO *label* [*loop-control*]
- R825 *nonlabel-do-stmt* **is** [*do-construct-name* :] DO [*loop-control*]
- R826 *loop-control* **is** [,] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■
■ [, *scalar-int-expr*]
or [,] WHILE (*scalar-logical-expr*)
or [,] CONCURRENT *forall-header*
- R827 *do-variable* **is** *scalar-int-variable-name*
- C815 (R827) The *do-variable* shall be a variable of type integer.
- R828 *do-block* **is** *block*
- R829 *end-do* **is** *end-do-stmt*
or *continue-stmt*
- R830 *end-do-stmt* **is** END DO [*do-construct-name*]
- C816 (R822) If the *do-stmt* of a *block-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 *block-do-construct* does not specify a *do-construct-name*, the corresponding *end-do* shall not specify a *do-construct-name*.
- C817 (R822) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.
- C818 (R822) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.
- R831 *nonblock-do-construct* **is** *action-term-do-construct*
or *outer-shared-do-construct*
- R832 *action-term-do-construct* **is** *label-do-stmt*
do-body
do-term-action-stmt
- R833 *do-body* **is** [*execution-part-construct*] ...
- R834 *do-term-action-stmt* **is** *action-stmt*
- C819 (R834) A *do-term-action-stmt* shall not be an *arithmetic-if-stmt*, *continue-stmt*, *cycle-stmt*, *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, *exit-stmt*, *goto-stmt*, *return-stmt*, or *stop-stmt*.
- C820 (R831) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer to the same label.
- R835 *outer-shared-do-construct* **is** *label-do-stmt*
do-body
shared-term-do-construct

- R836 *shared-term-do-construct* **is** *outer-shared-do-construct*
or *inner-shared-do-construct*
- R837 *inner-shared-do-construct* **is** *label-do-stmt*
do-body
do-term-shared-stmt
- R838 *do-term-shared-stmt* **is** *action-stmt*
- C821 (R838) A *do-term-shared-stmt* shall not be an *arithmetic-if-stmt*, *cycle-stmt*, *end-function-stmt*, *end-program-stmt*, *end-mp-subprogram-stmt*, *end-subroutine-stmt*, *exit-stmt*, *goto-stmt*, *return-stmt*, or *stop-stmt*.
- C822 (R836) The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmts* of the *inner-shared-do-construct* and *outer-shared-do-construct* shall refer to the same label.
- R839 *cycle-stmt* **is** CYCLE [*do-construct-name*]
- C823 (R839) If a *do-construct-name* appears, the CYCLE statement shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.
- C824 (R839) A *cycle-stmt* shall not appear within the range of a DO CONCURRENT construct if it belongs to an outer construct.
- C825 A RETURN statement shall not appear within a DO CONCURRENT construct.
- C826 A branch (8.2) within a DO CONCURRENT construct shall not have a branch target that is outside the construct.
- C827 A reference to a nonpure procedure shall not appear within a DO CONCURRENT construct.
- C828 A reference to the procedure IEEE.GET_FLAG, IEEE.SET_HALTING_MODE, or IEEE.GET_-HALTING_MODE from the intrinsic module IEEE.EXCEPTIONS, shall not appear within a DO CONCURRENT construct.
- R840 *if-construct* **is** *if-then-stmt*
block
[*else-if-stmt*
block] ...
[*else-stmt*
block]
end-if-stmt
- R841 *if-then-stmt* **is** [*if-construct-name* :] IF (*scalar-logical-expr*) THEN
- R842 *else-if-stmt* **is** ELSE IF (*scalar-logical-expr*) THEN [*if-construct-name*]
- R843 *else-stmt* **is** ELSE [*if-construct-name*]
- R844 *end-if-stmt* **is** END IF [*if-construct-name*]
- C829 (R840) 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*.
- R845 *if-stmt* **is** IF (*scalar-logical-expr*) *action-stmt*
- C830 (R845) The *action-stmt* in the *if-stmt* shall not be an *end-function-stmt*, *end-mp-subprogram-stmt*, *end-program-stmt*, *end-subroutine-stmt*, or *if-stmt*.
- R846 *select-type-construct* **is** *select-type-stmt*
[*type-guard-stmt*
block] ...
end-select-type-stmt
- R847 *select-type-stmt* **is** [*select-construct-name* :] SELECT TYPE ■

- ([*associate-name* =>] *selector*)
- C831 (R847) If *selector* is not a named *variable*, *associate-name* => shall appear.
- C832 (R847) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.6.7).
- C833 (R847) The *selector* in a *select-type-stmt* shall be polymorphic.
- R848 *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*]
- C834 (R848) The *type-spec* or *derived-type-spec* shall specify that each length type parameter is assumed.
- C835 (R848) The *type-spec* or *derived-type-spec* shall not specify a sequence derived type or a type with the BIND attribute.
- C836 (R848) If *selector* is not unlimited polymorphic, the *type-spec* or *derived-type-spec* shall specify an extension of the declared type of *selector*.
- C837 (R848) 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*.
- C838 (R848) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-guard-stmt*.
- R849 *end-select-type-stmt* **is** END SELECT [*select-construct-name*]
- C839 (R846) 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*.
- R850 *exit-stmt* **is** EXIT [*construct-name*]
- C840 If a *construct-name* appears, the EXIT statement shall be within that construct; otherwise, it shall be within the range of at least one *do-construct*.
- C841 An *exit-stmt* shall not belong to a DO CONCURRENT construct, nor shall it appear within the range of a DO CONCURRENT construct if it belongs to a construct that contains that DO CONCURRENT construct.
- R851 *goto-stmt* **is** GO TO *label*
- C842 (R851) The *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.
- R852 *computed-goto-stmt* **is** GO TO (*label-list*) [,] *scalar-int-expr*
- C843 (R852) Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.
- R853 *arithmetic-if-stmt* **is** IF (*scalar-numeric-expr*) *label* , *label* , *label*
- C844 (R853) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.
- C845 (R853) The *scalar-numeric-expr* shall not be of type complex.
- R854 *continue-stmt* **is** CONTINUE
- R855 *stop-stmt* **is** STOP [*stop-code*]
- R856 *stop-code* **is** *scalar-char-initialization-expr*
 or *scalar-int-initialization-expr*
- C846 (R856) The *scalar-char-initialization-expr* shall be of default kind.
- C847 (R856) The *scalar-int-initialization-expr* shall be of default kind.
- R857 *sync-all-stmt* **is** SYNC ALL [([*sync-stat-list*])]

- R858 *sync-stat* **is** *STAT = stat-variable*
or *ERRMSG = errmsg-variable*
- C848 No specifier shall appear more than once in a given *sync-stat-list*.
- R859 *sync-team-stmt* **is** *SYNC TEAM (image-team [, sync-stat-list])*
- R860 *image-team* **is** *scalar-variable*
- C849 The *image-team* shall be a scalar variable of type *IMAGE_TEAM* from the intrinsic module *ISO_FORTRAN_ENV*.
- R861 *sync-images-stmt* **is** *SYNC IMAGES (image-set [, sync-stat-list])*
- R862 *image-set* **is** *int-expr*
or ***
- C850 An *image-set* that is an *int-expr* shall be scalar or of rank one.
- R863 *notify-stmt* **is** *NOTIFY (image-set [, sync-stat-list])*
- R864 *query-stmt* **is** *QUERY (image-set [, query-spec-list])*
- R865 *query-spec* **is** *READY = scalar-logical-variable*
or *sync-stat*
- C851 (R864) No specifier shall appear more than once in a given *query-spec-list*.
- R866 *sync-memory-stmt* **is** *SYNC MEMORY [([sync-stat-list])]*

Clause 9:

- R901 *io-unit* **is** *file-unit-number*
or ***
or *internal-file-variable*
- R902 *file-unit-number* **is** *scalar-int-expr*
- R903 *internal-file-variable* **is** *char-variable*
- C901 (R903) The *char-variable* shall not be an array section with a vector subscript.
- C902 (R903) The *char-variable* shall be of type default character, ASCII character, or ISO 10646 character.
- R904 *open-stmt* **is** *OPEN (connect-spec-list)*
- R905 *connect-spec* **is** *[UNIT =] file-unit-number*
or *ACCESS = scalar-default-char-expr*
or *ACTION = scalar-default-char-expr*
or *ASYNCHRONOUS = scalar-default-char-expr*
or *BLANK = scalar-default-char-expr*
or *DECIMAL = scalar-default-char-expr*
or *DELIM = scalar-default-char-expr*
or *ENCODING = scalar-default-char-expr*
or *ERR = label*
or *FILE = file-name-expr*
or *FORM = scalar-default-char-expr*
or *IOMSG = iomsg-variable*
or *IOSTAT = scalar-int-variable*
or *NEWUNIT = scalar-int-variable*
or *PAD = scalar-default-char-expr*
or *POSITION = scalar-default-char-expr*
or *RECL = scalar-int-expr*
or *ROUND = scalar-default-char-expr*
or *SIGN = scalar-default-char-expr*

- or STATUS = *scalar-default-char-expr*
- or TEAM = *image-team*
- R906 *file-name-expr* is *scalar-default-char-expr*
- R907 *iomsg-variable* is *scalar-default-char-variable*
- C903 No specifier shall appear more than once in a given *connect-spec-list*.
- C904 (R904) If the NEWUNIT= specifier does not appear, a *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *connect-spec-list*.
- C905 (R904) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the OPEN statement.
- C906 (R904) If a NEWUNIT= specifier appears, a *file-unit-number* shall not appear.
- R908 *close-stmt* is CLOSE (*close-spec-list*)
- R909 *close-spec* is [UNIT =] *file-unit-number*
- or IOSTAT = *scalar-int-variable*
- or IOMSG = *iomsg-variable*
- or ERR = *label*
- or STATUS = *scalar-default-char-expr*
- C907 No specifier shall appear more than once in a given *close-spec-list*.
- C908 A *file-unit-number* shall be specified in a *close-spec-list*; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *close-spec-list*.
- C909 (R909) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.
- R910 *read-stmt* is READ (*io-control-spec-list*) [*input-item-list*]
- or READ *format* [, *input-item-list*]
- R911 *write-stmt* is WRITE (*io-control-spec-list*) [*output-item-list*]
- R912 *print-stmt* is PRINT *format* [, *output-item-list*]
- R913 *io-control-spec* is [UNIT =] *io-unit*
- or [FMT =] *format*
- or [NML =] *namelist-group-name*
- or ADVANCE = *scalar-default-char-expr*
- or ASYNCHRONOUS = *scalar-char-initialization-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 = *scalar-int-variable*
- or IOMSG = *iomsg-variable*
- or IOSTAT = *scalar-int-variable*
- or PAD = *scalar-default-char-expr*
- or POS = *scalar-int-expr*
- or REC = *scalar-int-expr*
- or ROUND = *scalar-default-char-expr*
- or SIGN = *scalar-default-char-expr*

or SIZE = *scalar-int-variable*

- C910 No specifier shall appear more than once in a given *io-control-spec-list*.
- C911 An *io-unit* shall be specified in an *io-control-spec-list*; if the optional characters UNIT= are omitted, the *io-unit* shall be the first item in the *io-control-spec-list*.
- C912 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.
- C913 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.
- C914 (R913) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the data transfer statement.
- C915 (R913) A *namelist-group-name* shall be the name of a namelist group.
- C916 (R913) A *namelist-group-name* shall not appear if an *input-item-list* or an *output-item-list* appears in the data transfer statement.
- C917 (R913) An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.
- C918 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.
- C919 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.
- C920 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC= specifier or a POS= specifier.
- C921 (R913) If the REC= specifier appears, an END= specifier shall not appear, a *namelist-group-name* shall not appear, and the *format*, if any, shall not be an asterisk.
- C922 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream input/output statement with explicit format specification (10.2) whose control information list does not contain an *internal-file-variable* as the *io-unit*.
- C923 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.
- C924 (R913) If a SIZE= specifier appears, an ADVANCE= specifier also shall appear.
- C925 (R913) The *scalar-char-initialization-expr* in an ASYNCHRONOUS= specifier shall be of type default character and shall have the value YES or NO.
- C926 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a *file-unit-number*.
- C927 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall also appear.
- C928 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.
- C929 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format* or *namelist-group-name* shall also appear.
- C930 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name* shall appear.
- R914 *format* **is** *default-char-expr*
 or *label*
 or *
- C931 (R914) The *label* shall be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the FMT= specifier.
- R915 *input-item* **is** *variable*
 or *io-implied-do*
- R916 *output-item* **is** *expr*
 or *io-implied-do*
- R917 *io-implied-do* **is** (*io-implied-do-object-list* , *io-implied-do-control*)
- R918 *io-implied-do-object* **is** *input-item*
 or *output-item*

- R919 *io-implied-do-control* **is** *do-variable = scalar-int-expr* , ■
 ■ *scalar-int-expr* [, *scalar-int-expr*]
- C932 (R915) A variable that is an *input-item* shall not be a whole assumed-size array.
- C933 (R915) A variable that is an *input-item* shall not be a procedure pointer.
- C934 (R919) The *do-variable* shall be a named scalar variable of type integer.
- C935 (R918) 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*.
- C936 (R916) An expression that is an *output-item* shall not have a value that is a procedure pointer.
- R920 *dtv-type-spec* **is** TYPE(*derived-type-spec*)
 or CLASS(*derived-type-spec*)
- C937 (R920) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used; otherwise, the TYPE keyword shall be used.
- C938 (R920) All length type parameters of *derived-type-spec* shall be assumed.
- R921 *wait-stmt* **is** WAIT (*wait-spec-list*)
- R922 *wait-spec* **is** [UNIT =] *file-unit-number*
 or END = *label*
 or EOR = *label*
 or ERR = *label*
 or ID = *scalar-int-expr*
 or IOMSG = *iomsg-variable*
 or IOSTAT = *scalar-int-variable*
- C939 No specifier shall appear more than once in a given *wait-spec-list*.
- C940 A *file-unit-number* shall be specified in a *wait-spec-list*; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *wait-spec-list*.
- C941 (R922) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the WAIT statement.
- R923 *backspace-stmt* **is** BACKSPACE *file-unit-number*
 or BACKSPACE (*position-spec-list*)
- R924 *endfile-stmt* **is** ENDFILE *file-unit-number*
 or ENDFILE (*position-spec-list*)
- R925 *rewind-stmt* **is** REWIND *file-unit-number*
 or REWIND (*position-spec-list*)
- R926 *position-spec* **is** [UNIT =] *file-unit-number*
 or IOMSG = *iomsg-variable*
 or IOSTAT = *scalar-int-variable*
 or ERR = *label*
- C942 No specifier shall appear more than once in a given *position-spec-list*.
- C943 A *file-unit-number* shall be specified in a *position-spec-list*; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *position-spec-list*.
- C944 (R926) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the file positioning statement.
- R927 *flush-stmt* **is** FLUSH *file-unit-number*
 or FLUSH (*flush-spec-list*)
- R928 *flush-spec* **is** [UNIT =] *file-unit-number*
 or IOSTAT = *scalar-int-variable*
 or IOMSG = *iomsg-variable*

- or** ERR = *label*
- C945 No specifier shall appear more than once in a given *flush-spec-list*.
- C946 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*.
- C947 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the FLUSH statement.
- R929 *inquire-stmt* **is** INQUIRE (*inquire-spec-list*)
 or INQUIRE (IOLENGTH = *scalar-int-variable*) ■
 ■ *output-item-list*
- R930 *inquire-spec* **is** [UNIT =] *file-unit-number*
 or FILE = *file-name-expr*
 or ACCESS = *scalar-default-char-variable*
 or ACTION = *scalar-default-char-variable*
 or ASYNCHRONOUS = *scalar-default-char-variable*
 or BLANK = *scalar-default-char-variable*
 or DECIMAL = *scalar-default-char-variable*
 or DELIM = *scalar-default-char-variable*
 or DIRECT = *scalar-default-char-variable*
 or ENCODING = *scalar-default-char-variable*
 or ERR = *label*
 or EXIST = *scalar-default-logical-variable*
 or FORM = *scalar-default-char-variable*
 or FORMATTED = *scalar-default-char-variable*
 or ID = *scalar-int-expr*
 or IOMSG = *iomsg-variable*
 or IOSTAT = *scalar-int-variable*
 or NAME = *scalar-default-char-variable*
 or NAMED = *scalar-default-logical-variable*
 or NEXTREC = *scalar-int-variable*
 or NUMBER = *scalar-int-variable*
 or OPENED = *scalar-default-logical-variable*
 or PAD = *scalar-default-char-variable*
 or PENDING = *scalar-default-logical-variable*
 or POS = *scalar-int-variable*
 or POSITION = *scalar-default-char-variable*
 or READ = *scalar-default-char-variable*
 or READWRITE = *scalar-default-char-variable*
 or RECL = *scalar-int-variable*
 or ROUND = *scalar-default-char-variable*
 or SEQUENTIAL = *scalar-default-char-variable*
 or SIGN = *scalar-default-char-variable*
 or SIZE = *scalar-int-variable*
 or STREAM = *scalar-default-char-variable*
 or TEAM = *image-team*
 or UNFORMATTED = *scalar-default-char-variable*

or WRITE = *scalar-default-char-variable*

C948 No specifier shall appear more than once in a given *inquire-spec-list*.

C949 An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not both.

C950 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*.

C951 If an ID= specifier appears in an *inquire-spec-list*, a PENDING= specifier shall also appear.

C952 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the INQUIRE statement.

Clause 10:

R1001 *format-stmt* **is** FORMAT *format-specification*

R1002 *format-specification* **is** ([*format-item-list*])
or ([*format-item-list*,] *unlimited-format-item*)

C1001 (R1001) The *format-stmt* shall be labeled.

C1002 (R1002) The comma used to separate *format-items* in a *format-item-list* may be omitted

R1003 *format-item* **is** [*r*] *data-edit-desc*
or *control-edit-desc*
or *char-string-edit-desc*
or [*r*] (*format-item-list*)

R1004 *unlimited-format-item* **is** * (*format-item-list*)

R1005 *r* **is** *int-literal-constant*

C1003 (R1005) *r* shall be positive.

C1004 (R1005) *r* shall not have a kind parameter specified for it.

R1006 *data-edit-desc* **is** I *w* [. *m*]
or B *w* [. *m*]
or O *w* [. *m*]
or Z *w* [. *m*]
or F *w* . *d*
or E *w* . *d* [E *e*]
or EN *w* . *d* [E *e*]
or ES *w* . *d* [E *e*]
or G *w* [. *d* [E *e*]]
or L *w*
or A [*w*]
or D *w* . *d*
or DT [*char-literal-constant*] [(*v-list*)]

R1007 *w* **is** *int-literal-constant*

R1008 *m* **is** *int-literal-constant*

R1009 *d* **is** *int-literal-constant*

R1010 *e* **is** *int-literal-constant*

R1011 *v* **is** *signed-int-literal-constant*

C1005 (R1010) *e* shall be positive.

C1006 (R1007) *w* shall be zero or positive for the I, B, O, Z, F, and G edit descriptors. *w* shall be positive for all other edit descriptors.

C1007 (R1006) For the G edit descriptor, *d* shall be specified if and only if *w* is not zero.

C1008 (R1006) *w*, *m*, *d*, *e*, and *v* shall not have kind parameters specified for them.

C1009 (R1006) The *char-literal-constant* in the DT edit descriptor shall not have a kind parameter

specified for it.

- R1012 *control-edit-desc* **is** *position-edit-desc*
 or [*r*] /
 or :
 or *sign-edit-desc*
 or *k P*
 or *blank-interp-edit-desc*
 or *round-edit-desc*
 or *decimal-edit-desc*
- R1013 *k* **is** *signed-int-literal-constant*
- C1010 (R1013) *k* shall not have a kind parameter specified for it.
- R1014 *position-edit-desc* **is** *T n*
 or *TL n*
 or *TR n*
 or *n X*
- R1015 *n* **is** *int-literal-constant*
- C1011 (R1015) *n* shall be positive.
- C1012 (R1015) *n* shall not have a kind parameter specified for it.
- R1016 *sign-edit-desc* **is** *SS*
 or *SP*
 or *S*
- R1017 *blank-interp-edit-desc* **is** *BN*
 or *BZ*
- R1018 *round-edit-desc* **is** *RU*
 or *RD*
 or *RZ*
 or *RN*
 or *RC*
 or *RP*
- R1019 *decimal-edit-desc* **is** *DC*
 or *DP*
- R1020 *char-string-edit-desc* **is** *char-literal-constant*
- C1013 (R1020) The *char-literal-constant* shall not have a kind parameter specified for it.
- R1021 *hex-digit-string* **is** *hex-digit* [*hex-digit*] ...

Clause 11:

- R1101 *main-program* **is** [*program-stmt*]
 [*specification-part*]
 [*execution-part*]
 [*internal-subprogram-part*]
 end-program-stmt
- R1102 *program-stmt* **is** *PROGRAM program-name*
- R1103 *end-program-stmt* **is** *END* [*PROGRAM* [*program-name*]]
- C1101 (R1101) In a *main-program*, the *execution-part* shall not contain a RETURN statement or an ENTRY statement.
- C1102 (R1101) The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, shall be identical to the *program-name* specified in the

- program-stmt.*
- R1104 *module* **is** *module-stmt*
 [*specification-part*]
 [*module-subprogram-part*]
 end-module-stmt
- R1105 *module-stmt* **is** **MODULE** *module-name*
- R1106 *end-module-stmt* **is** **END** [**MODULE** [*module-name*]]
- R1107 *module-subprogram-part* **is** *contains-stmt*
 [*module-subprogram*] ...
- R1108 *module-subprogram* **is** *function-subprogram*
 or *subroutine-subprogram*
 or *separate-module-subprogram*
- C1103 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* specified in the *module-stmt*.
- C1104 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.
- C1105 (R1104) If an object that has a default-initialized direct component is declared in the specification part of a module it shall have the **ALLOCATABLE**, **POINTER**, or **SAVE** attribute.
- C1106 If an object that has a co-array ultimate component is declared in the *specification-part* of a module it shall have the **SAVE** attribute.
- R1109 *use-stmt* **is** **USE** [[, *module-nature*] ::] *module-name* [, *rename-list*]
 or **USE** [[, *module-nature*] ::] *module-name* , ■
 ■ **ONLY** : [*only-list*]
- R1110 *module-nature* **is** **INTRINSIC**
 or **NON_INTRINSIC**
- R1111 *rename* **is** *local-name* => *use-name*
 or **OPERATOR** (*local-defined-operator*) => ■
 ■ **OPERATOR** (*use-defined-operator*)
- R1112 *only* **is** *generic-spec*
 or *only-use-name*
 or *rename*
- R1113 *only-use-name* **is** *use-name*
- C1107 (R1109) If *module-nature* is **INTRINSIC**, *module-name* shall be the name of an intrinsic module.
- C1108 (R1109) If *module-nature* is **NON_INTRINSIC**, *module-name* shall be the name of a nonintrinsic module.
- C1109 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the same name.
- C1110 (R1111) **OPERATOR**(*use-defined-operator*) shall not identify a type-bound generic interface.
- C1111 (R1112) The *generic-spec* shall not identify a type-bound generic interface.
- C1112 (R1112) Each *generic-spec* shall be a public entity in the module.
- C1113 (R1113) Each *use-name* shall be the name of a public entity in the module.
- R1114 *local-defined-operator* **is** *defined-unary-op*
 or *defined-binary-op*
- R1115 *use-defined-operator* **is** *defined-unary-op*
 or *defined-binary-op*
- C1114 (R1115) Each *use-defined-operator* shall be a public entity in the module.
- R1116 *submodule* **is** *submodule-stmt*

- [*specification-part*]
 [*module-subprogram-part*]
end-submodule-stmt
- R1117 *submodule-stmt* **is** SUBMODULE (*parent-identifier*) *submodule-name*
 R1118 *parent-identifier* **is** *ancestor-module-name* [: *parent-submodule-name*]
 R1119 *end-submodule-stmt* **is** END [SUBMODULE [*submodule-name*]]
 C1115 (R1116) A submodule *specification-part* shall not contain a *format-stmt*, *entry-stmt*, or *stmt-function-stmt*.
 C1116 (R1116) An object with a default-initialized direct component that is declared in the specification part of a submodule shall have the ALLOCATABLE, POINTER, or SAVE attribute.
 C1117 (R1118) The *ancestor-module-name* shall be the name of a nonintrinsic module; the *parent-submodule-name* shall be the name of a descendant of that module.
 C1118 (R1116) If a *submodule-name* appears in the *end-submodule-stmt*, it shall be identical to the one in the *submodule-stmt*.
 R1120 *block-data* **is** *block-data-stmt*
 [*specification-part*]
 end-block-data-stmt
 R1121 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]
 R1122 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]
 C1119 (R1120) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-data-stmt*.
 C1120 (R1120) A *block-data specification-part* shall contain only derived-type definitions and ASYNCHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRINSIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration statements.
 C1121 (R1120) A type declaration statement in a *block-data specification-part* shall not contain ALLOCATABLE, EXTERNAL, or BIND attribute specifiers.

Clause 12:

- R1201 *interface-block* **is** *interface-stmt*
 [*interface-specification*] ...
 end-interface-stmt
- R1202 *interface-specification* **is** *interface-body*
 or *procedure-stmt*
- R1203 *interface-stmt* **is** INTERFACE [*generic-spec*]
 or ABSTRACT INTERFACE
- R1204 *end-interface-stmt* **is** END INTERFACE [*generic-spec*]
- R1205 *interface-body* **is** *function-stmt*
 [*specification-part*]
 end-function-stmt
 or *subroutine-stmt*
 [*specification-part*]
 end-subroutine-stmt
- R1206 *procedure-stmt* **is** [MODULE] PROCEDURE [::] *procedure-name-list*
 R1207 *generic-spec* **is** *generic-name*
 or OPERATOR (*defined-operator*)
 or ASSIGNMENT (=)

- R1208 *dtio-generic-spec* **or** *dtio-generic-spec*
 is READ (FORMATTED)
 or READ (UNFORMATTED)
 or WRITE (FORMATTED)
 or WRITE (UNFORMATTED)
- C1201 (R1201) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure defined by that subprogram.
- C1202 (R1201) The *generic-spec* shall be included in the *end-interface-stmt* only if it is provided in the *interface-stmt*. If the *end-interface-stmt* includes *generic-name*, the *interface-stmt* shall specify the same *generic-name*. If the *end-interface-stmt* includes ASSIGNMENT(=), the *interface-stmt* shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *dtio-generic-spec*, the *interface-stmt* shall specify the same *dtio-generic-spec*. If the *end-interface-stmt* includes OPERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If one *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the corresponding operator <, <=, >, >=, ==, or /=.
- C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword that specifies an intrinsic type.
- C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.
- C1205 (R1205) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, and procedure arguments.
- C1206 (R1205) An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-function-stmt*.
- C1207 (R1206) A *procedure-name* shall have an explicit interface and shall refer to an accessible procedure pointer, external procedure, dummy procedure, or module procedure.
- C1208 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall be accessible in the current scope as a module procedure.
- C1209 (R1206) A *procedure-name* shall not specify a procedure that is specified previously in any *procedure-stmt* in any accessible interface with the same generic identifier.
- C1210 (R1205) A module procedure interface body shall not appear in an abstract interface block.
- R1209 *import-stmt* **is** IMPORT [[::] *import-name-list*
- C1211 (R1209) The IMPORT statement is allowed only in an *interface-body* that is not a module procedure interface body.
- C1212 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.
- R1210 *external-stmt* **is** EXTERNAL [[::] *external-name-list*
- R1211 *procedure-declaration-stmt* **is** PROCEDURE ([*proc-interface*]) ■
 ■ [[, *proc-attr-spec*] ... ::] *proc-decl-list*
- R1212 *proc-interface* **is** *interface-name*
 or *declaration-type-spec*
- R1213 *proc-attr-spec* **is** *access-spec*
 or *proc-language-binding-spec*
 or INTENT (*intent-spec*)
 or OPTIONAL
 or POINTER
 or SAVE
- R1214 *proc-decl* **is** *procedure-entity-name* [=> *proc-pointer-init*]
- R1215 *interface-name* **is** *name*
- R1216 *proc-pointer-init* **is** *null-init*
 or *initial-proc-target*

- R1217 *initial-proc-target* **is** *procedure-name*
- C1213 (R1215) 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 13.6 and not marked with a bullet (●).
- C1214 (R1215) The *name* shall not be the same as a keyword that specifies an intrinsic type.
- C1215 If a procedure entity has the INTENT attribute or SAVE attribute, it shall also have the POINTER attribute.
- C1216 (R1211) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall specify an external procedure.
- C1217 (R1214) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.
- C1218 (R1217) The *procedure-name* shall be the name of an initialization target.
- C1219 (R1211) If *proc-language-binding-spec* with a NAME= is specified, then *proc-decl-list* shall contain exactly one *proc-decl*, which shall neither have the POINTER attribute nor be a dummy procedure.
- C1220 (R1211) 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*.
- R1218 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*
- C1221 (R1218) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.
- R1219 *function-reference* **is** *procedure-designator* ([*actual-arg-spec-list*])
- C1222 (R1219) The *procedure-designator* shall designate a function.
- C1223 (R1219) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.
- R1220 *call-stmt* **is** CALL *procedure-designator* [([*actual-arg-spec-list*])]
- C1224 (R1220) The *procedure-designator* shall designate a subroutine.
- R1221 *procedure-designator* **is** *procedure-name*
or *proc-component-ref*
or *data-ref* % *binding-name*
- C1225 (R1221) A *procedure-name* shall be the name of a procedure or procedure pointer.
- C1226 (R1221) A *binding-name* shall be a binding name (4.5.5) of the declared type of *data-ref*.
- C1227 (R1221) If *data-ref* is an array, the referenced type-bound procedure shall have the PASS attribute.
- R1222 *actual-arg-spec* **is** [*keyword* =] *actual-arg*
- R1223 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *proc-component-ref*
or *alt-return-spec*
- R1224 *alt-return-spec* **is** * *label*
- C1228 (R1222) The *keyword* = shall not appear if the interface of the procedure is implicit in the scoping unit.
- C1229 (R1222) 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.
- C1230 (R1222) Each *keyword* shall be the name of a dummy argument in the explicit interface of the procedure.
- C1231 (R1223) A nonintrinsic elemental procedure shall not be used as an actual argument.
- C1232 (R1223) A *procedure-name* shall be the name of an external, internal, module, or dummy procedure, a specific intrinsic function listed in 13.6 and not marked with a bullet (●), or a procedure

pointer.

- C1233 (R1224) The *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

C1234 An actual argument that is a co-indexed object shall not correspond to a dummy argument that has either the ASYNCHRONOUS or VOLATILE attribute.

C1235 (R1223) If an actual argument is an array section or an assumed-shape array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an assumed-shape array that does not have the CONTIGUOUS attribute.

C1236 (R1223) If an actual argument is a pointer array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an assumed-shape array that does not have the CONTIGUOUS attribute or a pointer array.

R1225 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

R1226 *prefix-spec* **is** *declaration-type-spec*
or ELEMENTAL
or IMPURE
or MODULE
or PURE
or RECURSIVE

C1237 (R1225) A *prefix* shall contain at most one of each *prefix-spec*.

C1238 (R1225) A *prefix* shall not specify both PURE and IMPURE.

C1239 (R1225) A *prefix* shall not specify both ELEMENTAL and RECURSIVE.

C1240 (R1225) A *prefix* shall not specify ELEMENTAL if *proc-language-binding-spec* appears in the *function-stmt* or *subroutine-stmt*.

C1241 (R1225) MODULE shall appear only within the *function-stmt* or *subroutine-stmt* of a module subprogram or of an interface body that is declared in the scoping unit of a module or submodule.

C1242 (R1225) If MODULE appears within the *prefix* in a module subprogram, an accessible module procedure interface having the same name as the subprogram shall be declared in the module or submodule in which the subprogram is defined, or shall be declared in an ancestor of that program unit.

C1243 (R1225) If MODULE appears within the *prefix* in a module subprogram, the subprogram shall specify the same characteristics and dummy argument names as its corresponding (12.6.2.5) module procedure interface body.

C1244 (R1225) If MODULE appears within the *prefix* in 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.

C1245 (R1225) If MODULE appears within the *prefix* in a module subprogram, RECURSIVE shall appear if and only if RECURSIVE appears in the *prefix* in the corresponding module procedure interface body.

R1227 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1228 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [*suffix*]

C1246 (R1228) If RESULT appears, *result-name* shall not be the same as *function-name* and shall not be the same as the *entry-name* in any ENTRY statement in the subprogram.

C1247 (R1228) If RESULT appears, the *function-name* shall not appear in any specification statement

in the scoping unit of the function subprogram.

R1229 *proc-language-binding-spec* **is** *language-binding-spec*

C1248 (R1229) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the *function-stmt* or *subroutine-stmt* of an internal procedure, or of an interface body for an abstract interface or a dummy procedure.

C1249 (R1229) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy arguments shall be a nonoptional interoperable variable (15.3.5, 15.3.6) or a nonoptional interoperable procedure (15.3.7). If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable scalar variable.

R1230 *dummy-arg-name* **is** *name*

C1250 (R1230) A *dummy-arg-name* shall be the name of a dummy argument.

R1231 *suffix* **is** *proc-language-binding-spec* [*RESULT* (*result-name*)]
or *RESULT* (*result-name*) [*proc-language-binding-spec*]

R1232 *end-function-stmt* **is** *END* [*FUNCTION* [*function-name*]]

C1251 (R1227) An internal function subprogram shall not contain an *ENTRY* statement.

C1252 (R1227) An internal function subprogram shall not contain an *internal-subprogram-part*.

C1253 (R1232) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-name* specified in the *function-stmt*.

R1233 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-subroutine-stmt

R1234 *subroutine-stmt* **is** [*prefix*] *SUBROUTINE* *subroutine-name* ■
■ [([*dummy-arg-list*]) [*proc-language-binding-spec*]]

C1254 (R1234) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

R1235 *dummy-arg* **is** *dummy-arg-name*
or *

R1236 *end-subroutine-stmt* **is** *END* [*SUBROUTINE* [*subroutine-name*]]

C1255 (R1233) An internal subroutine subprogram shall not contain an *ENTRY* statement.

C1256 (R1233) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

C1257 (R1236) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the *subroutine-stmt*.

R1237 *separate-module-subprogram* **is** *mp-subprogram-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-mp-subprogram-stmt

R1238 *mp-subprogram-stmt* **is** *MODULE PROCEDURE* *procedure-name*

R1239 *end-mp-subprogram-stmt* **is** *END* [*PROCEDURE* [*procedure-name*]]

C1258 (R1237) The *procedure-name* shall be the same as the name of an accessible module procedure interface that is declared in the module or submodule in which the *separate-module-subprogram* is defined, or is declared in an ancestor of that program unit.

C1259 (R1239) If a *procedure-name* appears in the *end-mp-subprogram-stmt*, it shall be identical to the *procedure-name* in the *MODULE PROCEDURE* statement.

R1240 *entry-stmt* **is** *ENTRY* *entry-name* [([*dummy-arg-list*]) [*suffix*]]

C1260 (R1240) If *RESULT* appears, the *entry-name* shall not appear in any specification or type-

declaration statement in the scoping unit of the function program.

- C1261 (R1240) An *entry-stmt* shall appear only in an *external-subprogram* or a *module-subprogram* that does not define a separate module procedure. An *entry-stmt* shall not appear within an *executable-construct*.
- C1262 (R1240) RESULT shall appear only if the *entry-stmt* is in a function subprogram.
- C1263 (R1240) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function subprogram.
- C1264 (R1240) If RESULT appears, *result-name* shall not be the same as the *function-name* in the FUNCTION statement and shall not be the same as the *entry-name* in any ENTRY statement in the subprogram.
- R1241 *return-stmt* **is** RETURN [*scalar-int-expr*]
- C1265 (R1241) The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.
- C1266 (R1241) The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.
- R1242 *contains-stmt* **is** CONTAINS
- R1243 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*
- C1267 (R1243) The *primaries* of the *scalar-expr* shall be constants (literal and named), references to variables, references to functions and function dummy procedures, and intrinsic operations. If *scalar-expr* contains a reference to a function or a function dummy procedure, 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 or a function dummy procedure 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.
- C1268 (R1243) 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.
- C1269 (R1243) 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.
- C1270 (R1243) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.
- C1271 (R1243) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.
- C1272 (R1243) Each variable reference in *scalar-expr* may be either a reference to a dummy argument of the statement function or a reference to a variable accessible in the same scoping unit as the statement function statement.
- C1273 The *specification-part* of a pure function subprogram shall specify that all its nonpointer dummy data objects have INTENT(IN).
- C1274 The *specification-part* of a pure subroutine subprogram shall specify the intents of all its non-pointer dummy data objects.
- C1275 A local variable of a pure subprogram, or of a BLOCK construct within a pure subprogram, shall not have the SAVE attribute.
- C1276 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are pure.
- C1277 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.
- C1278 All internal subprograms in a pure subprogram shall be pure.
- C1279 In a pure subprogram any designator with a base object that is in common or accessed by host or use association, is a dummy argument of a pure function, is a dummy argument with INTENT (IN) of a pure subroutine, or an object that is storage associated with any such variable, shall not be used
- C1280 Any procedure referenced in a pure subprogram, including one referenced via a defined operation,

defined assignment, user-defined derived-type input/output, or finalization, shall be pure.

- C1281 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.
- C1282 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-number* or ***.
- C1283 A pure subprogram shall not contain a *stop-stmt*.
- C1284 A co-indexed object shall not appear in a variable definition context in a pure subprogram.
- C1285 A pure subprogram shall not contain an image control statement (8.5.1).
- C1286 All dummy arguments of an elemental procedure shall be scalar non-co-array dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.
- C1287 The result variable of an elemental function shall be scalar and shall not have the POINTER or ALLOCATABLE attribute.
- C1288 In the scoping unit of an elemental subprogram, an object designator with a dummy argument as the base object shall not appear in a *specification-expr* except as the designator in a type parameter inquiry (6.1.4) or as the argument to one of the intrinsic functions BIT_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

Clause 13:

Clause 14:

Clause 15:

- C1501 (R430) A derived type with the BIND attribute shall not be a SEQUENCE type.
- C1502 (R430) A derived type with the BIND attribute shall not have type parameters.
- C1503 (R430) A derived type with the BIND attribute shall not have the EXTENDS attribute.
- C1504 (R430) A derived type with the BIND attribute shall not have a *type-bound-procedure-part*.
- C1505 (R430) Each component of a derived type with the BIND attribute shall be a nonpointer, nonallocatable data component with interoperable type and type parameters.
- C1506 A procedure defined in a submodule shall not have a binding label unless its interface is declared in the ancestor module.

Clause 16:

E.2 Syntax rule cross-reference

R474	<i>ac-do-variable</i>	R473, C508
R472	<i>ac-implied-do</i>	R471, C508
R473	<i>ac-implied-do-control</i>	R472
R468	<i>ac-spec</i>	R467
R471	<i>ac-value</i>	R468, R472, C505, C506, C507
R525	<i>access-id</i>	R524, C562
R507	<i>access-spec</i>	R316, R432, R442, R446, R454, R455, R502, C517, R524, R1213
R524	<i>access-stmt</i>	R212, C562
R214	<i>action-stmt</i>	R213, C326, C327, R834, R838, R845, C830
R832	<i>action-term-do-construct</i>	R831
R1223	<i>actual-arg</i>	R1222
R1222	<i>actual-arg-spec</i>	C501, R1219, C1223, R1220, C1229
R709	<i>add-op</i>	R310, R706
R705	<i>add-operand</i>	R705, R706
R627	<i>alloc-opt</i>	R626, C636

R527	<i>allocatable-decl</i>	R526
R526	<i>allocatable-stmt</i>	R212
R636	<i>allocate-co-array-spec</i>	R631, C634, C635
R637	<i>allocate-co-shape-spec</i>	R636, C635
R632	<i>allocate-object</i>	R631, C628, C629, C630, C631, C632, C633, C634, C635, C638, C639, C642, R640, C644
R633	<i>allocate-shape-spec</i>	R631, C633, C635
R626	<i>allocate-stmt</i>	R214
R631	<i>allocation</i>	R626
R302	<i>alphanumeric-character</i>	R301, R304
R1224	<i>alt-return-spec</i>	C1223, R1223
—	<i>ancestor-module-name</i>	R1118, C1117
R719	<i>and-op</i>	R310, R715
R714	<i>and-operand</i>	R715
—	<i>arg-name</i>	R446, C459, R455, C479
R348	<i>arg-token</i>	R347
R853	<i>arithmetic-if-stmt</i>	R214, C327, C819, C821, C844
R467	<i>array-constructor</i>	C505, C506, C507, R701
R617	<i>array-element</i>	R536, C569, C572, R566, R603, R610
—	<i>array-name</i>	R543, R544
R618	<i>array-section</i>	R603
R510	<i>array-spec</i>	R502, R503, C515, R509, C532, R526, R527, R543, R544, R555, R556, R568, C597
R734	<i>assignment-stmt</i>	R214, R747, R757, C746
R336	<i>assignment-tok-sequence</i>	R335
R802	<i>associate-construct</i>	R213, C804
—	<i>associate-construct-name</i>	R803, R806, C804
—	<i>associate-name</i>	R804, C801, C802, R847, C831, C832
R803	<i>associate-stmt</i>	R802, C802, C804
R804	<i>association</i>	R803
R515	<i>assumed-shape-spec</i>	R510
R517	<i>assumed-size-spec</i>	R510
R528	<i>asynchronous-stmt</i>	R212
R502	<i>attr-spec</i>	R501, C501, C579
R923	<i>backspace-stmt</i>	R214, C1281
R346	<i>basic-token</i>	R345, R348
R345	<i>basic-token-sequence</i>	R341, R344, R345
R426	<i>binary-constant</i>	R425
R530	<i>bind-entity</i>	R529, C564
R529	<i>bind-stmt</i>	R212
R455	<i>binding-attr</i>	R453, C477, C480, C481
—	<i>binding-name</i>	R453, R454, C472, R1221, C1226
R451	<i>binding-private-stmt</i>	R450, C468
R425	<i>bits-literal-constant</i>	R306
R1017	<i>blank-interp-edit-desc</i>	R1012
R801	<i>block</i>	R802, R807, R810, R818, C814, R828, R840, R846
R807	<i>block-construct</i>	R213, C807

_____	<i>block-construct-name</i>	R808, R809, C807
R1120	<i>block-data</i>	R202, C1120, C1121
_____	<i>block-data-name</i>	R1121, R1122, C1119
R1121	<i>block-data-stmt</i>	R1120, C1119
R822	<i>block-do-construct</i>	R821, C816
R808	<i>block-stmt</i>	R807, C807
R738	<i>bounds-remapping</i>	R735, C720, C721
R737	<i>bounds-spec</i>	R735, C719
R1220	<i>call-stmt</i>	R214, C1233
R810	<i>case-construct</i>	R213, C808, C810, C812
_____	<i>case-construct-name</i>	R811, R812, R813, C808
R814	<i>case-expr</i>	R811, C810, C811, C812
R815	<i>case-selector</i>	R812
R812	<i>case-stmt</i>	R810, C808
R817	<i>case-value</i>	R816, C810
R816	<i>case-value-range</i>	R815, C811, C812
R309	<i>char-constant</i>	C303
R725	<i>char-expr</i>	C706, R731, R814
R731	<i>char-initialization-expr</i>	R508, C522, C713, R817, R856, C846, R913, C925
R422	<i>char-length</i>	R421, C419, C454, R503, C504
R423	<i>char-literal-constant</i>	R306, R1006, C1009, R1020, C1013
R420	<i>char-selector</i>	R404
R1020	<i>char-string-edit-desc</i>	R1003
R606	<i>char-variable</i>	C606, R903, C901, C902
R909	<i>close-spec</i>	R908, C907, C908
R908	<i>close-stmt</i>	R214, C1281
R511	<i>co-array-spec</i>	R442, C447, C448, R503, R509, C524, C536, C537, R527, R544
_____	<i>co-name</i>	R544
R625	<i>co-subscript</i>	C615, R624
_____	<i>common-block-name</i>	R530, R553, R567, C806
R568	<i>common-block-object</i>	R567, C597, C598, C599, C600
R567	<i>common-stmt</i>	R212
R417	<i>complex-literal-constant</i>	R306
R615	<i>complex-part-designator</i>	R603, R618, C624
R444	<i>component-array-spec</i>	R442, C446, C450
R442	<i>component-attr-spec</i>	R441, C443, C464
R461	<i>component-data-source</i>	R460
R443	<i>component-decl</i>	R441, C462
R440	<i>component-def-stmt</i>	R439, C443, C444, C445, C455
R447	<i>component-initialization</i>	C462, C463, C464
_____	<i>component-name</i>	C465
R439	<i>component-part</i>	R430
R460	<i>component-spec</i>	R459, C495, C496, C497, C498, C500, C501
R852	<i>computed-goto-stmt</i>	R214, C843
R711	<i>concat-op</i>	R310, R710
R905	<i>connect-spec</i>	R904, C903, C904

R305	<i>constant</i>	R308, R309, R540, R610, R701
R542	<i>constant-subobject</i>	R540, R541, C577
—	<i>construct-name</i>	R850, C840
R1242	<i>contains-stmt</i>	R210, R450, R1107
R854	<i>continue-stmt</i>	R214, C327, R829, C819
R1012	<i>control-edit-desc</i>	R1003
R818	<i>critical-construct</i>	R213, C813, C814
—	<i>critical-construct-name</i>	R819, R820, C813
R819	<i>critical-stmt</i>	R818, C813
R839	<i>cycle-stmt</i>	R214, C327, C819, C821, C824
R1009	<i>d</i>	R1006, C1007, C1008
R441	<i>data-component-def-stmt</i>	R440
R1006	<i>data-edit-desc</i>	R1003
R536	<i>data-i-do-object</i>	R535, C565, C567, C568, C572
R537	<i>data-i-do-variable</i>	R535, C572
R535	<i>data-implied-do</i>	R534, R536, C572
—	<i>data-pointer-component-name</i>	R736, C724
R736	<i>data-pointer-object</i>	R735, C717, C718, C719, C720, C721, C725
R612	<i>data-ref</i>	C612, C617, C618, R614, R617, R618, C723, C729, R1221, C1226, C1227
R532	<i>data-stmt</i>	R209, R212, C1206
R540	<i>data-stmt-constant</i>	R538, C575
R534	<i>data-stmt-object</i>	R533, C565, C566, C567, C568
R539	<i>data-stmt-repeat</i>	R538, C573
R533	<i>data-stmt-set</i>	R532
R538	<i>data-stmt-value</i>	R533
R739	<i>data-target</i>	R461, C502, C503, C552, R735, C717, C718, C721, C727
R641	<i>dealloc-opt</i>	R640, C645
R640	<i>deallocate-stmt</i>	R214
R1019	<i>decimal-edit-desc</i>	R1012
R207	<i>declaration-construct</i>	R204
R403	<i>declaration-type-spec</i>	C404, C405, C423, R441, C444, C445, R501, R560, R1212, R1226, C1254
R726	<i>default-char-expr</i>	C707, R905, R906, R909, R913, R914
R607	<i>default-char-variable</i>	C607, R629, R907, R930
R605	<i>default-logical-variable</i>	C605, R930
R519	<i>deferred-co-shape-spec</i>	C447, C524, R511, C536
R516	<i>deferred-shape-spec</i>	R442, R444, C446, R510, C532, R550
R315	<i>define-macro-stmt</i>	R314
R723	<i>defined-binary-op</i>	R311, R722, C704, R1114, R1115
R311	<i>defined-operator</i>	C474, R1207, C1202
R703	<i>defined-unary-op</i>	R311, R702, C703, R1114, R1115
R430	<i>derived-type-def</i>	R207, C437, C441, C442
R457	<i>derived-type-spec</i>	R402, C403, R403, C405, C406, R459, C494, C501, R848, C834, C835, C836, R920, C937, C938
R431	<i>derived-type-stmt</i>	R430, C431, C438, C441, C442

R603	<i>designator</i>	R448, C466, R542, R601, C601, R615, C621, R616, C622, R701, C702, C746
—	<i>digit</i>	R302, R313, R410, R426, C412, R427, C413, R429, R426, C428, R427, C429, R429
R410	<i>digit-string</i>	R407, R408, R409, R413, R414
R544	<i>dimension-decl</i>	R543
R543	<i>dimension-stmt</i>	R212
R828	<i>do-block</i>	R822
R833	<i>do-body</i>	R832, R835, R837
R821	<i>do-construct</i>	R213, C823, C840
—	<i>do-construct-name</i>	R824, R825, R830, C816, R839, C823
R823	<i>do-stmt</i>	R822, C816, C817, C818
R834	<i>do-term-action-stmt</i>	C327, R832, C819, C820
R838	<i>do-term-shared-stmt</i>	R837, C821, C822
R827	<i>do-variable</i>	R474, R537, R826, C815, R919, C934
R1208	<i>dtio-generic-spec</i>	C476, R1207, C1202
R1235	<i>dummy-arg</i>	R1234, R1240, C1263
R1230	<i>dummy-arg-name</i>	R545, R546, R557, R1228, C1250, R1235, R1243, C1269, C1270, C1271
R1010	<i>e</i>	R1006, C1005, C1008
R842	<i>else-if-stmt</i>	R840, C829
R843	<i>else-stmt</i>	R840, C829
R750	<i>elsewhere-stmt</i>	R744, C735
R806	<i>end-associate-stmt</i>	R802, C804
R1122	<i>end-block-data-stmt</i>	R1120, C1119
R809	<i>end-block-stmt</i>	R807, C807
R820	<i>end-critical-stmt</i>	R818, C813
R829	<i>end-do</i>	R822, C816, C817, C818
R830	<i>end-do-stmt</i>	R829, C816, C817
R466	<i>end-enum-stmt</i>	R462
R758	<i>end-forall-stmt</i>	R752, C737
R1232	<i>end-function-stmt</i>	R214, C201, C326, C327, C819, C821, C830, R1205, R1227, C1253
R844	<i>end-if-stmt</i>	R840, C829
R1204	<i>end-interface-stmt</i>	R1201, C1202
R340	<i>end-macro-stmt</i>	R314
R1106	<i>end-module-stmt</i>	R1104, C1103
R1239	<i>end-mp-subprogram-stmt</i>	R214, C201, C326, C327, C819, C821, C830, R1237, C1259
R1103	<i>end-program-stmt</i>	R214, C201, C326, C327, C819, C821, C830, R1101, C1102
R813	<i>end-select-stmt</i>	R810, C808
R849	<i>end-select-type-stmt</i>	R846, C839
R1119	<i>end-submodule-stmt</i>	R1116, C1118
R1236	<i>end-subroutine-stmt</i>	R214, C201, C326, C327, C819, C821, C830, R1205, R1233, C1257
R434	<i>end-type-stmt</i>	R430
R751	<i>end-where-stmt</i>	R744, C735
R924	<i>endfile-stmt</i>	R214, C1281
R503	<i>entity-decl</i>	R501, C502, C503, C505

——	<i>entity-name</i>	R530, R551
——	<i>entry-name</i>	C1246, R1240, C1260, C1264
R1240	<i>entry-stmt</i>	R206, R207, R209, C1104, C1115, C1206, C1261, C1262
R462	<i>enum-def</i>	R207
R463	<i>enum-def-stmt</i>	R462
R465	<i>enumerator</i>	R464, C504
R464	<i>enumerator-def-stmt</i>	R462
R721	<i>equiv-op</i>	R310, R717
R716	<i>equiv-operand</i>	R716, R717
R566	<i>equivalence-object</i>	R565, C585, C586, C587, C588, C589, C590, C591, C592, C593, C594, C595
R565	<i>equivalence-set</i>	R564
R564	<i>equivalence-stmt</i>	R212
R629	<i>errmsg-variable</i>	R627, R641, R858
R213	<i>executable-construct</i>	R208, R209, C1261
R208	<i>execution-part</i>	C201, R1101, C1101, R1227, R1233, R1237
R209	<i>execution-part-construct</i>	R208, R801, R833
R850	<i>exit-stmt</i>	R214, C327, C819, C821, C841
R342	<i>expand-stmt</i>	R323, C332
R520	<i>explicit-co-shape-spec</i>	R511, C537
R512	<i>explicit-shape-spec</i>	R444, C450, C451, C515, R510, C531, R517, C597
R416	<i>exponent</i>	R413
R415	<i>exponent-letter</i>	R413, C414
R722	<i>expr</i>	R461, R471, R601, C602, R630, R701, R722, R724, R725, R726, R727, R728, R730, R734, R739, C728, R742, C731, R805, C803, R916, R1223, R1243, C1267, C1268, C1272
R312	<i>extended-intrinsic-op</i>	R311
——	<i>external-name</i>	R1210
R1210	<i>external-stmt</i>	R212
R203	<i>external-subprogram</i>	R202, C1261
R906	<i>file-name-expr</i>	R905, R930
R902	<i>file-unit-number</i>	R901, R905, C904, C906, R909, C908, C920, C926, R922, C940, R923, R924, R925, R926, C943, R927, R928, C946, R930, C950, C1282
R456	<i>final-procedure-stmt</i>	R452
——	<i>final-subroutine-name</i>	R456, C485, C486
R928	<i>flush-spec</i>	R927, C945, C946
R927	<i>flush-stmt</i>	R214, C1281
R757	<i>forall-assignment-stmt</i>	R756, C746, R759
R756	<i>forall-body-construct</i>	R752, C743, C744, C745
R752	<i>forall-construct</i>	R213, R756, C743
——	<i>forall-construct-name</i>	R753, R758, C737
R753	<i>forall-construct-stmt</i>	R752, C737
R754	<i>forall-header</i>	R753, R759, R826
R759	<i>forall-stmt</i>	R214, R756
R755	<i>forall-triplet-spec</i>	R754, C742
R914	<i>format</i>	R910, R912, R913, C917, C918, C921, C929, C930

R1003	<i>format-item</i>	R1002, C1002, R1003, R1004
R1002	<i>format-specification</i>	R1001
R1001	<i>format-stmt</i>	R206, R207, R209, C1001, C1104, C1115, C1206
—	<i>function-name</i>	R503, C508, C1203, R1228, C1246, C1247, R1232, C1253, C1264, R1243, C1270
R1219	<i>function-reference</i>	R506, C512, R701
R1228	<i>function-stmt</i>	R1205, C1203, C1240, C1241, R1227, C1248, C1253
R1227	<i>function-subprogram</i>	R203, R211, R1108
—	<i>generic-name</i>	C473, R1207, C1202
R1207	<i>generic-spec</i>	R454, C471, C473, C474, C475, C476, R525, R1112, C1111, C1112, R1203, R1204, C1202, C1204
R851	<i>goto-stmt</i>	R214, C327, C819, C821, C842
R428	<i>hex-constant</i>	R425
R429	<i>hex-digit</i>	R428, R1021
R840	<i>if-construct</i>	R213, C829
—	<i>if-construct-name</i>	R841, R842, R843, R844, C829
R845	<i>if-stmt</i>	R214, C326, C830
R841	<i>if-then-stmt</i>	R840, C829
R419	<i>imag-part</i>	R417
R624	<i>image-selector</i>	R613, C615, C616, C617
R862	<i>image-set</i>	C850, R863, R864
R860	<i>image-team</i>	C849, R905, R930
R205	<i>implicit-part</i>	R204
R206	<i>implicit-part-stmt</i>	R205
R560	<i>implicit-spec</i>	R559
R559	<i>implicit-stmt</i>	R205, R206
R518	<i>implied-shape-spec</i>	R510
—	<i>import-name</i>	R1209, C1212
R1209	<i>import-stmt</i>	R204
—	<i>index-name</i>	R755, C741, C742, C743
R448	<i>initial-data-target</i>	R447, C465, R505, C511, R540
R1217	<i>initial-proc-target</i>	R1216
R505	<i>initialization</i>	R503, C505, C506, C507, C510
R730	<i>initialization-expr</i>	R447, R505, R548, C712
R837	<i>inner-shared-do-construct</i>	R836, C822
R915	<i>input-item</i>	R910, C916, R918, C932, C933, C935
R930	<i>inquire-spec</i>	R929, C948, C949, C950, C951
R929	<i>inquire-stmt</i>	R214, C1281
R308	<i>int-constant</i>	C302, R539
—	<i>int-constant-name</i>	R408, C409
R541	<i>int-constant-subobject</i>	R539, C576
R727	<i>int-expr</i>	R401, R473, R611, R619, R622, R623, R625, R634, R635, C708, R729, C711, R732, R814, R826, R852, R862, C850, R902, R905, R913, R919, R922, R930, R1241, C1266
R732	<i>int-initialization-expr</i>	R405, C408, R420, C417, R437, R465, R535, C714, R817, R856, C847
R407	<i>int-literal-constant</i>	R306, R406, R422, C418, R1005, R1007, R1008, R1009, R1010, R1015

R608	<i>int-variable</i>	C608, R628, R905, R909, R913, R922, R926, R928, R929, R930
—	<i>int-variable-name</i>	R827
R523	<i>intent-spec</i>	R502, R545, R1213
R545	<i>intent-stmt</i>	R212
R1201	<i>interface-block</i>	R207, C1201
R1205	<i>interface-body</i>	R1202, C1201, C1205, C1206, C1211
R1215	<i>interface-name</i>	R453, C482, R1212, C1220
R1202	<i>interface-specification</i>	R1201
R1203	<i>interface-stmt</i>	R1201, C1202, C1203
R903	<i>internal-file-variable</i>	R901, C922
R211	<i>internal-subprogram</i>	R210
R210	<i>internal-subprogram-part</i>	R1101, R1227, C1252, R1233, C1256, R1237
R310	<i>intrinsic-operator</i>	R312, C703, C704
—	<i>intrinsic-procedure-name</i>	R1218, C1221
R1218	<i>intrinsic-stmt</i>	R212
R404	<i>intrinsic-type-spec</i>	R402, R403
R913	<i>io-control-spec</i>	R910, R911, C910, C911, C917, C918, C919, C920, C928
R917	<i>io-implied-do</i>	R915, R916
R919	<i>io-implied-do-control</i>	R917
R918	<i>io-implied-do-object</i>	R917, C935
R901	<i>io-unit</i>	R913, C911, C918, C919, C920, C922, C926, C1282
R907	<i>iomsg-variable</i>	R905, R909, R913, R922, R926, R928, R930
R1013	<i>k</i>	R1012, C1010
R215	<i>keyword</i>	R458, C491, C492, R460, C498, C499, R1222, C1228, C1229, C1230
R408	<i>kind-param</i>	R407, C410, C411, R413, C414, C415, C418, R423, C426, R424, C427, R425
R405	<i>kind-selector</i>	R404, R436
R313	<i>label</i>	C304, R824, C818, R851, C842, R852, C843, R853, C844, R905, C905, R909, C909, R913, C914, R914, C931, R922, C941, R926, C944, R928, C947, R930, C952, R1224, C1233
R824	<i>label-do-stmt</i>	R823, C818, R832, C820, R835, R837, C822
R508	<i>language-binding-spec</i>	R502, C502, C503, R529, C564, R1229
R469	<i>lbracket</i>	R347, R442, R467, R503, R509, R527, R544, R624, R631
R421	<i>length-selector</i>	R420, C423, C424
—	<i>letter</i>	R302, R304, R561, C581, R703, R723
R561	<i>letter-spec</i>	R560
R702	<i>level-1-expr</i>	R704
R706	<i>level-2-expr</i>	R706, R710
R710	<i>level-3-expr</i>	R710, R712
R712	<i>level-4-expr</i>	R714
R717	<i>level-5-expr</i>	R717, R722
R306	<i>literal-constant</i>	R305
R1114	<i>local-defined-operator</i>	R1111
—	<i>local-name</i>	R1111
R724	<i>logical-expr</i>	C705, R733, R748, R814, R826, R841, R842, R845
R733	<i>logical-initialization-expr</i>	C715, R817

R424	<i>logical-literal-constant</i>	R306, C703, C704
R604	<i>logical-variable</i>	C604
R826	<i>loop-control</i>	R824, R825
R513	<i>lower-bound</i>	R512, R515, R517, R518
R634	<i>lower-bound-expr</i>	R633, R636, R637, R737, R738
R521	<i>lower-co-bound</i>	R520, C538
R1008	<i>m</i>	R1006, C1008
R343	<i>macro-actual-arg</i>	C329, C330, C332
R322	<i>macro-body-block</i>	R314, R324, R328
R323	<i>macro-body-construct</i>	R322, C309
R337	<i>macro-body-stmt</i>	R323, C317, C318
R333	<i>macro-condition</i>	R329, R330
R317	<i>macro-declaration-stmt</i>	R314
R314	<i>macro-definition</i>	R207, R323, C309
R324	<i>macro-do-construct</i>	R323
R326	<i>macro-do-limit</i>	C311
R325	<i>macro-do-stmt</i>	R324
—	<i>macro-do-variable-name</i>	C310
—	<i>macro-dummy-arg-name</i>	C305, C307
—	<i>macro-dummy-arg-name-list</i>	C305
—	<i>macro-dummy-name</i>	C331, C332
R330	<i>macro-else-if-stmt</i>	R328
R331	<i>macro-else-stmt</i>	R328
R327	<i>macro-end-do-stmt</i>	R324
R332	<i>macro-end-if-stmt</i>	R328
R341	<i>macro-expr</i>	R321, C308, R326, R333, R334, C320
R328	<i>macro-if-construct</i>	R323
R329	<i>macro-if-then-stmt</i>	R328
R334	<i>macro-int-assignment-stmt</i>	R323
—	<i>macro-integer-variable-name</i>	R334, C313
—	<i>macro-local-variable-name</i>	C306
—	<i>macro-local-variable-name-list</i>	R320
—	<i>macro-name</i>	R340, C319, C321
R319	<i>macro-optional-decl-stmt</i>	R317
R335	<i>macro-tok-assignment-stmt</i>	R323
—	<i>macro-tok-variable-name</i>	R335, C314
R318	<i>macro-type-declaration-stmt</i>	R317
R320	<i>macro-variable-decl-stmt</i>	R317, C306
R1101	<i>main-program</i>	R202, C1101
R748	<i>mask-expr</i>	R743, R745, R749, R754, C739, C740
R749	<i>masked-elsewhere-stmt</i>	R744, C735
R1104	<i>module</i>	R202
—	<i>module-name</i>	R1105, R1106, C1103, R1109, C1107, C1108
R1110	<i>module-nature</i>	R1109, C1107, C1108
R1105	<i>module-stmt</i>	R1104, C1103
R1108	<i>module-subprogram</i>	R1107, C1261

R1107	<i>module-subprogram-part</i>	R1104, R1116
R1238	<i>mp-subprogram-stmt</i>	R1237
R708	<i>mult-op</i>	R310, R705
R704	<i>mult-operand</i>	R704, R705
R1015	<i>n</i>	R1014, C1011, C1012
R304	<i>name</i>	R102, R215, C301, R307, R504, R554, R602, R1215, C1213, C1214, R1230
R307	<i>named-constant</i>	R305, R418, R419, R465, R548
R548	<i>named-constant-def</i>	R547
—	<i>namelist-group-name</i>	R562, C582, C584, R913, C915, C916, C917, C919, C921, C929, C930
R563	<i>namelist-group-object</i>	R562, C583, C584
R562	<i>namelist-stmt</i>	R212
R347	<i>nested-token-sequence</i>	R345
R831	<i>nonblock-do-construct</i>	R821
R825	<i>nonlabel-do-stmt</i>	R823, C817
R718	<i>not-op</i>	R310, R714
R863	<i>notify-stmt</i>	R214
R506	<i>null-init</i>	R447, R505, R540, R1216
R638	<i>nullify-stmt</i>	R214
R728	<i>numeric-expr</i>	C709, R853, C845
R504	<i>object-name</i>	R503, C506, C509, C511, R526, R527, R528, R531, R550, R553, R555, R556, R558, R603
R427	<i>octal-constant</i>	R425
R1112	<i>only</i>	R1109
R1113	<i>only-use-name</i>	R1112
R904	<i>open-stmt</i>	R214, C1281
R546	<i>optional-stmt</i>	R212
R720	<i>or-op</i>	R310, R716
R715	<i>or-operand</i>	R715, R716
R835	<i>outer-shared-do-construct</i>	R831, R836, C822
R916	<i>output-item</i>	R911, R912, C916, R918, C935, C936, R929
R547	<i>parameter-stmt</i>	R206, R207
R1118	<i>parent-identifier</i>	R1117
R610	<i>parent-string</i>	R609, C609
—	<i>parent-submodule-name</i>	R1118, C1117
—	<i>parent-type-name</i>	R432, C432
—	<i>part-name</i>	R613, C610, C611, C612, C613, C614, C615, C616, C619, C620, C625
R613	<i>part-ref</i>	C568, C571, C586, R612, C619, C620, C623, C624
R735	<i>pointer-assignment-stmt</i>	R214, C552, R757
R550	<i>pointer-decl</i>	R549
R639	<i>pointer-object</i>	R638, C643
R549	<i>pointer-stmt</i>	R212
R1014	<i>position-edit-desc</i>	R1012
R926	<i>position-spec</i>	R923, R924, R925, C942, C943
R707	<i>power-op</i>	R310, R704

R1225	<i>prefix</i>	C1237, C1238, C1239, C1240, C1242, C1243, C1244, C1245, R1228, R1234, C1254
R1226	<i>prefix-spec</i>	R1225, C1237
—	<i>primaries</i>	C1267
R701	<i>primary</i>	R702
R912	<i>print-stmt</i>	R214, C1281
R449	<i>private-components-stmt</i>	R433, C467
R433	<i>private-or-sequence</i>	R430, C437
R1213	<i>proc-attr-spec</i>	R1211
R446	<i>proc-component-attr-spec</i>	R445, C456, C457, C460
R445	<i>proc-component-def-stmt</i>	R440, C456
R741	<i>proc-component-ref</i>	R740, R742, R1221, R1223
R1214	<i>proc-decl</i>	R445, R1211, C1217, C1219
—	<i>proc-entity-name</i>	R550
R1212	<i>proc-interface</i>	R445, R1211, C1216, C1220
R1229	<i>proc-language-binding-spec</i>	C516, R1213, C1219, C1220, C1240, C1248, C1249, R1231, R1234
R1216	<i>proc-pointer-init</i>	R1214
R554	<i>proc-pointer-name</i>	R553, R568, C598, C601, R639, R740
R740	<i>proc-pointer-object</i>	R735
R742	<i>proc-target</i>	R461, C502, C552, R735, C733
—	<i>procedure-component-name</i>	C730
R1211	<i>procedure-declaration-stmt</i>	R207, C1213
R1221	<i>procedure-designator</i>	R1219, C1222, R1220, C1224
—	<i>procedure-entity-name</i>	R1214, C1216
—	<i>procedure-name</i>	R453, C469, C470, R742, C732, R1206, C1207, C1208, C1209, R1217, C1218, R1221, C1225, R1223, C1232, R1238, R1239, C1258, C1259
R1206	<i>procedure-stmt</i>	R1202, C1204, C1208, C1209
—	<i>program-name</i>	R1102, R1103, C1102
R1102	<i>program-stmt</i>	R1101, C1102
R202	<i>program-unit</i>	R201
R551	<i>protected-stmt</i>	R212
R865	<i>query-spec</i>	R864, C851
R864	<i>query-stmt</i>	R214
R1005	<i>r</i>	R1003, C1003, C1004, R1012
R470	<i>rbracket</i>	R347, R442, R467, R503, R509, R527, R544, R624, R631
R910	<i>read-stmt</i>	R214, C912, C1282
R413	<i>real-literal-constant</i>	R306, R412
R418	<i>real-part</i>	R417
R713	<i>rel-op</i>	R310, R712
R1111	<i>rename</i>	R1109, R1112
—	<i>rep-char</i>	R423
—	<i>result-name</i>	C1246, R1231, C1264
R338	<i>result-token</i>	R336, R337, C315
R1241	<i>return-stmt</i>	R214, C327, C819, C821, C1265
R925	<i>rewind-stmt</i>	R214, C1281

R1018	<i>round-edit-desc</i>	R1012
R552	<i>save-stmt</i>	R212
R553	<i>saved-entity</i>	R552, C806
R103	<i>scalar-xyz</i>	C101
R620	<i>section-subscript</i>	R613, C614, C616, C624
R811	<i>select-case-stmt</i>	R810, C808
—	<i>select-construct-name</i>	R847, R848, R849, C839
R846	<i>select-type-construct</i>	R213, C837, C838, C839
R847	<i>select-type-stmt</i>	R846, C833, C839
R805	<i>selector</i>	R804, C801, R847, C831, C832, C833, C836
R1237	<i>separate-module-subprogram</i>	R1108, C1258
R435	<i>sequence-stmt</i>	R433
R836	<i>shared-term-do-construct</i>	R835
R411	<i>sign</i>	R406, R409, R412
R1016	<i>sign-edit-desc</i>	R1012
R409	<i>signed-digit-string</i>	R416
R406	<i>signed-int-literal-constant</i>	R418, R419, R540, R1011, R1013
R412	<i>signed-real-literal-constant</i>	R418, R419, R540
R414	<i>significand</i>	R413
R630	<i>source-expr</i>	R627, C629, C633, C637, C638, C639, C641
—	<i>special-character</i>	R301
R729	<i>specification-expr</i>	C404, C419, R513, R514, R521, R522, C1288
R204	<i>specification-part</i>	C471, C517, C562, R807, C805, R1101, R1104, C1104, C1106, R1116, C1115, R1120, C1120, C1121, R1205, R1227, R1233, R1237, C1273, C1274, C1276
R212	<i>specification-stmt</i>	R207
R628	<i>stat-variable</i>	R627, R641, R858
R1243	<i>stmt-function-stmt</i>	R207, C1104, C1115, C1206
R856	<i>stop-code</i>	R855
R855	<i>stop-stmt</i>	R214, C327, C819, C821, C1283
R622	<i>stride</i>	R621, R755, C742
R614	<i>structure-component</i>	R536, C570, C571, C572, R603, R610, R632, R639
R459	<i>structure-constructor</i>	R540, C575, R701
R1116	<i>submodule</i>	R202
—	<i>submodule-name</i>	R1117, R1119, C1118
R1117	<i>submodule-stmt</i>	R1116, C1118
—	<i>subroutine-name</i>	C1203, R1234, R1236, C1257
R1234	<i>subroutine-stmt</i>	R1205, C1203, C1240, C1241, C1248, R1233, C1254, C1257
R1233	<i>subroutine-subprogram</i>	R203, R211, R1108
R619	<i>subscript</i>	C571, C623, R620, R621, R755, C742
R621	<i>subscript-triplet</i>	R620, C627
R609	<i>substring</i>	R566, C596, R603
R611	<i>substring-range</i>	R609, R618, C625
R1231	<i>suffix</i>	R1228, R1240
R857	<i>sync-all-stmt</i>	R214
R861	<i>sync-images-stmt</i>	R214

R866	<i>sync-memory-stmt</i>	R214
R858	<i>sync-stat</i>	R857, C848, R863, R865, R866
R859	<i>sync-team-stmt</i>	R214
R556	<i>target-decl</i>	R555
R555	<i>target-stmt</i>	R212
R339	<i>token</i>	R338
R432	<i>type-attr-spec</i>	R431, C431
R454	<i>type-bound-generic-stmt</i>	R452, C471
R452	<i>type-bound-proc-binding</i>	R450
R450	<i>type-bound-procedure-part</i>	R430, C440, C1504
R453	<i>type-bound-procedure-stmt</i>	R452
R501	<i>type-declaration-stmt</i>	R207, C423, C424, C501
R848	<i>type-guard-stmt</i>	R846, C837, C838, C839
—	<i>type-name</i>	R431, C430, R434, C438, C476, R457, C488
R438	<i>type-param-attr-spec</i>	R436
R437	<i>type-param-decl</i>	R436
R436	<i>type-param-def-stmt</i>	R430, C441, C442
R616	<i>type-param-inquiry</i>	R701
—	<i>type-param-name</i>	R431, R437, C441, C442, R616, C622, R701, C701
R458	<i>type-param-spec</i>	R457, C489, C490, C491, C493
R401	<i>type-param-value</i>	C401, C402, C404, R420, R421, R422, C419, C420, C421, C422, C455, R458, C493, C631
R402	<i>type-spec</i>	R468, C505, C506, C507, R626, C629, C630, C631, C632, C637, C640, R754, C738, R848, C834, C835, C836
R303	<i>underscore</i>	R302
R1004	<i>unlimited-format-item</i>	R1002
R514	<i>upper-bound</i>	R512
R635	<i>upper-bound-expr</i>	R633, R637, R738
R522	<i>upper-co-bound</i>	R520, C538
R1115	<i>use-defined-operator</i>	R1111, C1110, C1114
—	<i>use-name</i>	R525, C563, R1111, R1113, C1113
R1109	<i>use-stmt</i>	R204
R1011	<i>v</i>	R1006, C1008
R557	<i>value-stmt</i>	R212
R601	<i>variable</i>	R534, C566, C568, R604, R605, R606, R607, R608, R734, C716, R736, C723, C724, R739, C726, C729, C730, C746, R805, C801, C831, C832, R915, R1223
R602	<i>variable-name</i>	R563, R566, R568, C598, C601, C603, R610, R632, R639, R736, C722
R623	<i>vector-subscript</i>	R620, C626
R558	<i>volatile-stmt</i>	R212
R1007	<i>w</i>	R1006, C1006, C1007, C1008
R922	<i>wait-spec</i>	R921, C939, C940
R921	<i>wait-stmt</i>	R214, C1281
R747	<i>where-assignment-stmt</i>	R743, R746, C734
R746	<i>where-body-construct</i>	R744, C736
R744	<i>where-construct</i>	R213, R746, R756
—	<i>where-construct-name</i>	R745, R749, R750, R751, C735

R745 *where-construct-stmt*
R743 *where-stmt*
R911 *write-stmt*

R744, C735
R214, R746, R756
R214, C913, C1282

Annex F

(Informative)

Index

In this annex, entries in *italics* denote BNF terms, entries in **bold face** denote language keywords, and page numbers in **bold face** denote primary text or glossary definitions.

Symbols

<, 148
<=, 148
>, 148
>=, 148
(, 147
, 33, 44, 46, 47, 52, 93, 95, 103, 126, 143, 233, 265, 278, 284, 311, 332
**, 143
+, 143
-, 143
.AND., 146, 147
.EQ., 148
.EQV., 146, 147
.GE., 148
.GT., 148
.LE., 148
.LT., 148
.NE., 148
.NEQV., 146, 147
.NOT., 146, 147
.OR., 146, 147
.XOR., 146, 147
, 143
//, 54, 145
/=, 148
;, 33
==, 148
&, 32, 282

A

abstract interface, **299**
abstract interface block, **301**
abstract type, **74**, **505**
ac-do-variable (R474), 82, **82**, 155, 157, 486, 487
ac-implied-do (R472), 82, **82**, 139, 487
ac-implied-do-control (R473), 82, **82**, 139, 154–157
ac-spec (R468), 82, **82**
ac-value (R471), 82, **82**
access methods, **208**
access-id (R525), 101, **101**
access-spec (R507), 35, 58, 63, 64, 69–72, 85, 88, **88**, 101, 308

access-stmt (R524), 11, 101, **101**
ACCESS= specifier, 218, 249
accessibility attribute, **88**, 101
accessibility statement, **101**
action-stmt (R214), 11, **11**, 38, 185, 191, 195
action-term-do-construct (R832), 185, **185**
ACTION= specifier, 218, 249
actions, **208**
active, **186**
active combination, **172**
actual argument, **297**, **505**
actual-arg (R1223), 311, **311**
actual-arg-spec (R1222), 78, 310, 311, **311**
add-op (R709), 28, 134, **134**, 135
add-operand (R705), 134, **134**, 135, 138
ADVANCE= specifier, 226
advancing input/output statement, **212**
affector, **227**
alloc-opt (R627), 125, 126, **126**
allocatable array, **92**
ALLOCATABLE attribute, **88**, 102
ALLOCATABLE statement, **102**
allocatable variable, **505**
allocatable-decl (R527), 102, **102**
allocatable-stmt (R526), 11, **102**, 488
ALLOCATE statement, **125**
allocate-co-array-spec (R636), 126, **126**, 128
allocate-co-shape-spec (R637), 126, **126**
allocate-object (R632), 52, 53, 126, **126**, 127, 128, 130, 132, 503
allocate-shape-spec (R633), 126, **126**, 127, 128
allocate-stmt (R626), 11, **125**, 503
allocated, **129**
allocation (R631), 125, **126**, 128
alphanumeric-character (R302), 25, **25**, 27
alt-return-spec (R1224), 195, 311, **311**
ancestor, **294**
ancestor component, **75**
ancestor-module-name, 294
and-op (R719), 28, 136, **136**
and-operand (R714), 136, **136**
approximation methods, **48**
arg-name, 64, 66, 71

- arg-token* (R348), 38, **38**
 - argument, **505**
 - argument association, 313, 487, **505**
 - argument keyword, **21**, 313, 341, 486
 - arithmetic IF statement, **195**
 - arithmetic-if-stmt* (R853), 12, 38, 185, 195, **195**
 - array, **19**, 90–94, **121**, 121–124, **505**
 - assumed-shape, 92
 - assumed-size, 93
 - deferred-shape, 92
 - explicit-shape, 92
 - array constructor, 81, **81**
 - array element, **19**, **121**, 122, **505**
 - array element order, **123**
 - array intrinsic assignment statement, **159**
 - array pointer, **93**, **505**
 - array section, **19**, **123**, **505**
 - array-constructor* (R467), 82, **82**, 133
 - array-element* (R617), 103, 111, 117, 118, **122**
 - array-name*, 105, 488
 - array-section* (R618), 117, 122, **122**, 123
 - array-spec* (R510), 7, 86, 88, 90, 91, **91**, 93, 102, 105, 107, 113, 114, 128
 - ASCII character set, **51**
 - ASCII character type, **51**, 158, 214, 231, 264, 279, 410, 422
 - ASCII collating sequence, **55**, 352, 386, 389, 396–398, 410
 - assignment, 157–175
 - defined, 162
 - elemental array (FORALL), 169
 - intrinsic, 158
 - masked array (WHERE), 166
 - pointer, 163
 - assignment statement, 157, **505**
 - assignment-stmt* (R734), 11, 158, **158**, 167, 169, 170, 172, 338, 503
 - assignment-tok-sequence* (R336), 36, **37**
 - ASSOCIATE construct, 178, **178**, 487
 - associate name, **178**, **505**
 - ASSOCIATE statement, 490
 - associate-construct* (R802), 11, 178, **178**
 - associate-construct-name*, 178
 - associate-name*, 178, 192–194, 486, 515
 - associate-stmt* (R803), 178, **178**, 195
 - associated, **20**
 - associating entity, **497**
 - association, **22**, **505**
 - argument, 313, 487
 - common, 114
 - host, 488
 - inheritance, 497
 - name, 487
 - pointer, 491
 - sequence, 321
 - storage, 493
 - use, 488
 - association* (R804), 178, **178**
 - association status, pointer, 491
 - assumed type parameter, **45**
 - assumed-shape array, **92**, **505**
 - assumed-shape-spec* (R515), 91, 92, **92**
 - assumed-size array, **93**, **505**
 - assumed-size-spec* (R517), 91, **93**
 - asynchronous, 219, 222, **226**, 227, 229, 231–233, 240, 243–245, 247, 249, 251, 252
 - ASYNCHRONOUS attribute, **88**, 102
 - ASYNCHRONOUS statement, **102**
 - asynchronous-stmt* (R528), 11, **102**
 - ASYNCHRONOUS= specifier**, 219, 226, 249
 - attr-spec* (R502), 85, **85**, 86, 87, 107
 - attribute, **85**, 87–101, **505**
 - accessibility, **88**, 101
 - ALLOCATABLE, **88**, 102
 - ASYNCHRONOUS, **88**, 102
 - BIND, 57, 60, 102
 - CONTIGUOUS, **89**, 102, 319
 - DIMENSION, **90**, 105
 - EXTERNAL, **95**, 302, 307, 308
 - INTENT, **95**, 105
 - INTRINSIC, **97**, 310
 - OPTIONAL, **97**, 106
 - PARAMETER, **98**, 106
 - POINTER, **98**, 106
 - PRIVATE, 58, **88**, 101
 - PROTECTED, **98**, 106
 - PUBLIC, 58, **88**, 101
 - SAVE, 103, 107
 - TARGET, **100**, 107, 315
 - VALUE, **100**, 107, 314
 - VOLATILE, **100**, 107
 - attribute specification statements, 101–116
 - automatic data object, **86**, **505**
- ## B
- BACKSPACE statement, **245**
 - backspace-stmt* (R923), 11, **244**, 338
 - base object, **119**
 - base type, **74**, **506**
 - basic-token* (R346), 38, **38**
 - basic-token-sequence* (R345), 37, 38, **38**
 - belong, **187**, **194**, **506**
 - belongs, **194**
 - binary-constant* (R426), 56, **56**
 - BIND attribute, 57, 60, 102
 - BIND statement, **102**
 - BIND(C)**, 80, 89, 298, 300, 475, 477, 481
 - bind-entity* (R530), 102, **102**

bind-stmt (R529), 11, **102**
 binding, **71**, 485
 binding label, **480**, **481**, **506**
 binding name, **71**
binding-attr (R455), 70, 71, **71**
binding-name, 70, 71, 311, 328, 486, 516
binding-private-stmt (R451), 70, **70**, 72
 bit model, 342
 bits compatible, **314**, **506**
 bits conversion, 161
 bits editing, 265
 bits intrinsic assignment statement, **158**
 bits intrinsic operation, **140**, 147
 bits intrinsic operator, **140**
 bits relational intrinsic operation, **140**
 bits type, **55**
bits-literal-constant (R425), 28, **56**
 blank common, **113**
blank-interp-edit-desc (R1017), 261, **262**
BLANK= specifier, 219, 227, 249
 block, **177**, **506**
block (R801), **177**, 178–180, 183, 185, 190, 192
 BLOCK construct, **179**
 block data program unit, **294**, **506**
 BLOCK DATA statement, **294**
 BLOCK statement, **179**
block-construct (R807), 11, **179**, 180
block-construct-name, 179, 180
block-data (R1120), 9, **294**, 295
block-data-name, 294, 295
block-data-stmt (R1121), 10, 294, **294**
block-do-construct (R822), 184, **184**, 185
block-stmt (R808), 179, **179**, 180, 195
 blocking, **203**
 bounds, **506**
bounds-remapping (R738), 163, **164**, 165
bounds-spec (R737), 163, **164**, 165
 branch target statement, **195**
 branching, **195**

C

C address, **468**
 C character kind, **468**
 C_(C type), 467–478
 C_LOC function, 471
 CALL statement, **311**
call-stmt (R1220), 11, 311, **311**, 314
 CASE construct, **180**
 case index, **181**
case-construct (R810), 11, 180, **180**, 181
case-construct-name, 180
case-expr (R814), 180, **180**, 181
case-selector (R815), 180, **180**
case-stmt (R812), 180, **180**

case-value (R817), 180, 181, **181**
case-value-range (R816), 180, **180**, 181
 CHAR intrinsic, 54
char-constant (R309), 28, **28**
char-expr (R725), 152, **152**, 157, 180
char-initialization-expr (R731), 89, 157, **157**, 181, 196, 224, 225
char-length (R422), 52, **52**, 53, 63, 64, 85–87
char-literal-constant (R423), 28, 34, **53**, 239, 261, 262, 593
char-selector (R420), 47, 52, **52**
char-string-edit-desc (R1020), 260, **262**
char-variable (R606), 117, **117**, 214
CHARACTER, 51
character (R301), **25**
 character context, **30**
 character intrinsic assignment statement, **158**
 character intrinsic operation, **140**
 character intrinsic operator, **140**
 character length parameter, **506**
 character literal constant, **53**
 character relational intrinsic operation, **140**
 character sequence type, **59**, 111, 496, 618
 character set, 25
 character storage unit, **494**, **506**
 character string, **51**, **506**
 character string edit descriptor, **260**
 character type, **51**, 51–55
 CHARACTER_STORAGE_SIZE, **436**
 characteristics, **506**
 characteristics of a procedure, **298**
 child, **294**
 child data transfer statement, **236**, 236–240, 257
CLASS, 46
 class, **506**
 CLASS DEFAULT statement, **192**
 CLASS IS statement, **192**
 CLOSE statement, **222**
close-spec (R909), 222, **222**
close-stmt (R908), 11, **222**, 338
 co-array, **20**, **506**
 explicit-co-shape, 95
co-array-spec (R511), 63, 64, 86, 90, 91, **91**, 94, 95, 102, 105, 107
 co-bound, **20**
 co-bounds, 94, 95, **507**
 co-dimension, **20**
 co-indexed object, **20**, **507**
co-name, 105
 co-rank, **20**, **507**
 co-size, **20**
 co-subscript, **507**
co-subscript (R625), 20, 119, 125, **125**
 collating sequence, 54, **54**, **507**

- collective subroutine, **341, 507**
 - comment, **32, 33**
 - common association, **114**
 - common block, **113**, 485, **507**, 552
 - common block storage sequence, **114**
 - COMMON statement, 113–116
 - common-block-name*, 102, 107, 113, 180, 293
 - common-block-object* (R568), 113, **113**, 293, 488
 - common-stmt* (R567), 11, **113**, 488
 - companion processor, **23, 507**
 - compatibility
 - FORTRAN 77, 4
 - Fortran 2003, 3
 - Fortran 90, 4
 - Fortran 95, 3
 - COMPILER_OPTIONS, **436**
 - COMPILER_VERSION, **436**
 - COMPLEX**, 50
 - complex part designator, **120**
 - complex type, 50, **50**
 - complex-literal-constant* (R417), 28, **50**
 - complex-part-designator* (R615), 117, 120, **120**, 122
 - component, **62**, 485, **507**
 - component definition statement, **62**
 - component keyword, **21**
 - component order, **69, 507**
 - component value, **77**
 - component-array-spec* (R444), 63, **63**, 64
 - component-attr-spec* (R442), 62, 63, **63**, 65, 66
 - component-data-source* (R461), 77, **77**, 78, 79
 - component-decl* (R443), 52, 62, **63**, 64, 66
 - component-def-stmt* (R440), 62, **62**, 63, 64
 - component-initialization* (R447), 63, 66, **66**
 - component-name*, 63, 67
 - component-part* (R439), 57, **62**, 69, 72
 - component-spec* (R460), 77, **77**, 78, 156
 - computed GO TO statement, **195**
 - computed-goto-stmt* (R852), 12, 195, **195**
 - concat-op* (R711), 28, 135, **135**
 - concatenation, **54**
 - conform, 158
 - conformable, **19, 507**
 - conformance, 154, **507**
 - connect team, **221, 223, 507**
 - connect-spec* (R905), 217, **217**, 218, 221
 - connected, **216, 507**
 - constant, **18**, 28, 43, **507**
 - character, 53
 - integer, 48
 - named, 106
 - constant* (R305), 28, **28**, 104, 118, 133
 - constant subobject, **19**
 - constant-subobject* (R542), 104, **104**
 - construct, **507**
 - construct association, 490, **507**
 - construct entity, **483, 507**
 - construct-name*, 194
 - constructor
 - array, 81
 - derived-type, 77
 - structure, 77
 - CONTAINS statement, 70, **336**
 - contains-stmt* (R1242), 10, 70, 290, **336**
 - contiguous, **89**
 - CONTIGUOUS attribute, **89**, 102, 319
 - CONTIGUOUS statement, **102**
 - contiguous-stmt* (R531), **102**
 - continuation, 32, 33
 - CONTINUE statement, **196**
 - continue-stmt* (R854), 11, 38, 185, **196**
 - control character, **25**
 - control edit descriptor, **260**, 274
 - control information list, **224**
 - control mask, **508**
 - control-edit-desc* (R1012), 260, **261**
 - conversion
 - bits, 161
 - numeric, 160
 - correspond, **334**
 - CRITICAL construct, **183**
 - CRITICAL statement, **183**
 - critical-construct* (R818), 11, 183, **183**
 - critical-construct-name*, 183
 - critical-stmt* (R819), 183, **183**, 195
 - current record, **211**
 - CYCLE statement, 184, 187
 - cycle-stmt* (R839), 11, 38, 185, 187, **187**
- ## D
- d* (R1009), 261, **261**, 266–269, 272, 273, 281
 - data, **508**
 - data edit descriptor, **260**, 264
 - data edit descriptors, 274
 - data entity, **18, 508**
 - data object, **18, 508**
 - data object reference, **22**
 - data pointer, **20**
 - DATA statement, **102**, 498
 - data transfer, 233
 - data transfer input statement, **207**
 - data transfer output statements, **207**
 - data transfer statements, 223
 - data type, *see* type, **508**
 - data-component-def-stmt* (R441), 62, **62**, 64
 - data-edit-desc* (R1006), 260, **261**
 - data-i-do-object* (R536), 103, **103**, 104
 - data-i-do-variable* (R537), 103, **103**, 104, 157, 486, 487

- data-implied-do* (R535), 103, **103**, 104, 157, 487
- data-pointer-component-name*, 163
- data-pointer-initialization compatible, **66**
- data-pointer-object* (R736), 163, **163**, 164, 165, 172, 356, 503
- data-ref* (R612), **118**, 119, 120, 122, 163, 164, 227, 311, 314, 320, 328
- data-stmt* (R532), 10, 11, **102**, 301, 337, 488
- data-stmt-constant* (R540), 103, 104, **104**
- data-stmt-object* (R534), 103, **103**, 104
- data-stmt-repeat* (R539), 103, **103**, 104
- data-stmt-set* (R533), 102, **103**
- data-stmt-value* (R538), 103, **103**, 104
- data-target* (R739), 77–79, 99, 163, 164, **164**, 165, 172, 322, 338, 356, 493
- datum, **508**
- dealloc-opt* (R641), 130, **130**, 131
- DEALLOCATE statement, **130**
- deallocate-stmt* (R640), 11, **130**, 503
- decimal symbol, **264**, **508**
- decimal-edit-desc* (R1019), 261, **262**
- DECIMAL= specifier**, 219, 227, 249
- declaration, **22**, 85–116
- declaration-construct* (R207), 10, **10**
- declaration-type-spec* (R403), 45, **45**, 46, 52, 62, 63, 85, 87, 108, 154, 308, 309, 329, 332
- declared type, **46**, **508**
- default bits, **56**
- default character, **51**
- default complex, **50**
- default initialization, **66**, **508**
- default integer, **47**
- default logical, **55**
- default real, **49**
- default-char-expr* (R726), 152, **152**, 217–228
- default-char-variable* (R607), 117, **117**, 126, 218, 248–254
- default-initialized, **67**, **508**
- default-logical-variable* (R605), 117, **117**, 248, 250, 251
- deferred binding, **71**, **508**
- deferred type parameter, **44**, **508**
- deferred-co-shape-spec* (R519), 63, 91, 94, **94**
- deferred-shape array, **92**
- deferred-shape-spec* (R516), 63, 91, 93, **93**, 106
- definable, **508**
- define-macro-stmt* (R315), 35, **35**, 488
- defined, **22**, **508**
- defined assignment, 305
- defined assignment statement, **162**, **508**
- defined binary operation, **150**
- defined elemental assignment statement, **162**
- defined elemental operation, **151**
- defined operation, **150**, 304, **508**
- defined operations, 150
- defined unary operation, **150**
- defined-binary-op* (R723), 29, 136, **136**, 137, 150, 151, 292
- defined-operator* (R311), **29**, 71, 292, 300, 301
- defined-unary-op* (R703), 29, 134, **134**, 137, 150, 151, 292
- definition, **22**
- definition of variables, 497
- deleted feature, **508**
- deleted features, **7**
- DELIM= specifier**, 219, 227, 250
- delimiters, **30**
- derived type, **17**, 57–80, **508**
- derived type determination, 60
- derived-type intrinsic assignment statement, **158**
- derived-type type specifier, 46
- derived-type-def* (R430), 10, 46, **57**, 58, 61
- derived-type-spec* (R457), 45, 52, 77, **77**, 78, 192, 237, 486
- derived-type-stmt* (R431), 57, **57**, 58, 61, 488
- descendant, **294**
- designator, **21**, **509**
- designator* (R603), 66, 67, 104, 117, **117**, 120, 121, 133, 170
- designator*, 133
- digit*, 6, 25, **25**, 29, 48, 56, 279
- digit-string* (R410), 48, **48**, 49, 266, 271
- digit-string*, 48
- digits, **25**
- DIMENSION attribute, **90**, 105
- DIMENSION statement, **105**
- dimension-decl* (R544), 105, **105**
- dimension-spec* (R509), **91**
- dimension-stmt* (R543), 11, **105**, 488
- direct access, 210
- direct access input/output statement, **228**
- direct component, **57**
- DIRECT= specifier**, 250
- disassociated, **20**, **509**
- distinguishable, **306**
- DO CONCURRENT construct, **184**
- DO CONCURRENT statement, 184
- DO construct, **184**
- DO statement, **184**
- DO termination, **185**
- DO WHILE statement, 184
- do-block* (R828), 184, **185**, 186
- do-body* (R833), 185, **185**, 186
- do-construct* (R821), 11, **184**, 187, 194
- do-construct-name*, 184, 185, 187
- do-stmt* (R823), 184, **184**, 185, 195, 503
- do-term-action-stmt* (R834), 38, 185, **185**, 187, 195

- do-term-shared-stmt* (R838), 185, **185**, 186, 187, 195
- do-variable* (R827), 82, 103, 184, **184**, 185, 186, 229, 230, 255–257, 280, 499, 501, 503, 544
- DOUBLE PRECISION**, 49
- double precision real, **49**
- dtio-generic-spec* (R1208), 71, 236–238, 242, 300, 301, **301**, 304, 307
- dtv-type-spec* (R920), **237**
- dummy argument, **297**, **509**
 - restrictions, 322
- dummy array, **509**
- dummy data object, **509**
- dummy procedure, **298**, **509**
- dummy-arg* (R1235), 332, **332**, 334
- dummy-arg-name* (R1230), 105–107, 330, **330**, 331, 332, 336, 488
- dynamic type, **46**, **509**
- E**
- e* (R1010), 261, **261**, 267–269, 272–274, 281
- edit descriptor, *see* format descriptor, **260**
- effective argument, **314**
- effective item, **230**, **509**
- effective position, **307**
- element sequence, **321**
- ELEMENTAL**, 329
- elemental, **20**, **297**, **509**
- elemental array assignment (FORALL), 169
- elemental intrinsic function, **341**
- elemental operation, **154**
- elemental procedure, **339**
- else-if-stmt* (R842), 190, **190**
- else-stmt* (R843), 190, **190**
- elsewhere-stmt* (R750), 167, **167**
- ENCODING= specifier**, 219, 250
- END BLOCK statement, **179**
- END CRITICAL statement, **183**
- END INTERFACE statement, **300**
- END statement, 16, **16**
- end-associate-stmt* (R806), 178, **178**, 195
- end-block-data-stmt* (R1122), 10, 16, 294, **294**
- end-block-stmt* (R809), 179, **179**, 180, 195
- end-critical-stmt* (R820), 183, **183**, 195
- end-do* (R829), 184, 185, **185**, 186–188
- end-do-stmt* (R830), 185, **185**, 195
- end-enum-stmt* (R466), 80, **80**
- end-forall-stmt* (R758), 169, **169**
- end-function-stmt* (R1232), 9, 11, 12, 16, 38, 185, 191, 300, 330, 331, **331**, 336
- end-if-stmt* (R844), 190, **190**, 195
- end-interface-stmt* (R1204), 300, **300**, 301
- end-macro-stmt* (R340), 35, **37**
- end-module-stmt* (R1106), 9, 16, 290, **290**
- end-mp-subprogram-stmt* (R1239), 11, 12, 16, 38, 185, 191, 333, **333**, 336
- end-of-file condition, **254**
- end-of-record condition, **254**
- end-program-stmt* (R1103), 9, 11, 12, 16, 17, 38, 185, 191, 289, **289**
- end-select-stmt* (R813), 180, **180**, 181, 195
- end-select-type-stmt* (R849), 192, **192**, 193, 195
- end-submodule-stmt* (R1119), 10, 16, 294, **294**
- end-subroutine-stmt* (R1236), 9, 11, 12, 16, 38, 185, 191, 300, 332, **332**, 333, 336
- end-type-stmt* (R434), 57, **58**
- end-where-stmt* (R751), 167, **167**
- END= specifier**, 255
- endfile record, **208**
- ENDFILE statement, 208, **245**
- endfile-stmt* (R924), 11, **244**, 338
- ending point, **118**
- entity, **509**
- entity-decl* (R503), 52, 85, 86, **86**, 87, 156, 157, 488
- entity-name*, 102, 106
- ENTRY statement, **334**
- entry-name*, 330, 334, 485
- entry-stmt* (R1240), 10, 290, 294, 301, 334, **334**, 485, 488
- enum-def* (R462), 10, 80, **80**, 81
- enum-def-stmt* (R463), 80, **80**
- enumeration, 80
- enumerator, 80
- enumerator* (R465), 80, **80**
- enumerator-def-stmt* (R464), 80, **80**
- EOR= specifier**, 255
- equiv-op* (R721), 28, 136, **136**
- equiv-operand* (R716), 136, **136**
- EQUIVALENCE statement, 110–113
- equivalence-object* (R566), 111, **111**, 112, 293
- equivalence-set* (R565), 110, **111**, 112
- equivalence-stmt* (R564), 11, **110**, 488
- ERR= specifier**, 255
- errmsg-variable* (R629), 126, **126**, 127, 130, 132, 199, 503, 594
- ERRMSG= specifier**, 132
- ERROR_UNIT, 215, **436**
- evaluation
 - operations, 139
 - optional, 151
 - parentheses, 152
- executable construct, 177, **509**
- executable statement, **15**, **509**
- executable-construct* (R213), 10, **11**, 15, 334
- execution control, 177
- execution cycle, **187**
- execution-part* (R208), 9, 10, **10**, 12, 289, 330–333
- execution-part-construct* (R209), 10, **10**, 177, 185

exist, **209**, **215**
EXIST= specifier, **250**
 EXIT statement, **194**
exit-stmt (R850), 11, 38, 185, **194**, 195
expand-stmt (R342), 35, **37**, 38
 explicit, **299**
 explicit formatting, 259–278
 explicit initialization, **87**, 102, **509**
 explicit interface, 299, **509**
 explicit-co-shape co-array, **95**
explicit-co-shape-spec (R520), 91, 95, **95**
 explicit-shape array, **92**, **509**
explicit-shape-spec (R512), 63, 64, 88, 91, 92, **92**,
 93, 94, 113
exponent (R416), 49, **50**
exponent-letter (R415), 49, **49**, 50
expr (R722), 7, 74, 77, 78, 82, 117, 126, 133, 134,
 136, **136**, 137, 152, 153, 157–164, 168,
 169, 172, 178, 229, 311, 336, 338, 462
 expression, **19**, **133**, 133–157, **509**
 extended type, **74**, **509**
extended-intrinsic-op (R312), 29, **29**
 extensible type, **74**, **509**
 extension operation, 137, **151**
 extension operator, **151**
 extension type, **74**, **510**
 extent, **19**, **510**
 EXTERNAL attribute, **95**, 302, 307, 308
 external file, **208**, **510**
 external linkage, **510**
 external procedure, **13**, **297**, **510**
 EXTERNAL statement, **307**
 external subprogram, **12**, **510**
 external unit, **214**, **510**
external-name, 307
external-stmt (R1210), 11, **307**, 488
external-subprogram (R203), 9, **9**, 334

F

field, **262**
 field width, **262**
 file, **510**
 file access, 209
 file connection, 214
 file connection statements, **207**
 file inquiry statement, **207**, 247
 file position, 211
 file positioning statements, **207**, 244
 file storage unit, **213**, **510**
file-name-expr (R906), 217, **218**, 219, 248, 249, 251
file-unit-number (R902), 214, **214**, 215, 217, 218,
 222, 224, 225, 238, 244–246, 248, 249, 338,
 438
FILE= specifier, 219, 249

FILE_STORAGE_SIZE, **436**
 files
 connected, 216
 external, 208
 internal, 214
 preconnected, 216
 FINAL statement, **72**
 final subroutine, **73**, **510**
final-procedure-stmt (R456), 70, **72**
final-subroutine-name, 72
 finalizable, **73**, **510**
 finalization, 73, **510**
 finalized, **73**
 fixed source form, 33, **33**
 FLUSH statement, **246**
flush-spec (R928), 246, **246**
flush-stmt (R927), 11, **246**, 338
FMT= specifier, 225
 FORALL construct, 169
forall-assignment-stmt (R757), 169, **169**, 170, 172,
 174, 338
forall-body-construct (R756), 169, **169**, 170, 172,
 174
forall-construct (R752), 11, 169, **169**, 170, 171, 173
forall-construct-name, 169, 170
forall-construct-stmt (R753), 169, **169**, 170, 195
forall-header (R754), 169, **169**, 174, 175, 184, 487
forall-stmt (R759), 11, 169, 173, 174, **174**
forall-triplet-spec (R755), 169, **169**, 170, 171, 173
FORM= specifier, 220, 250
format (R914), 223–225, **225**, 226, 233, 259, 260
 format control, **262**
 format descriptor
 /, 276
 .;, 276
 A, 272
 BN, 277
 BZ, 277
 control edit descriptor, 274
 D, 267
 data edit descriptor, 264–274
 E, 267
 EN, 268
 ES, 269
 F, 266
 G, 272
 I, 265
 L, 271
 P, 276
 S, 276
 SP, 276
 SS, 276
 TL, 275
 TR, 275

X, 275
 format descriptors
 G, 272
 FORMAT statement, 226, **259**, 632
format-item (R1003), 259, 260, **260**
format-specification (R1002), 259, **259**
format-stmt (R1001), 10, 259, **259**, 290, 294, 301
 formatted data transfer, 235
 formatted input/output statement, **225**
 formatted record, **207**
FORMATTED= specifier, 250
 formatting
 explicit, 259–278
 list-directed, 235, 278–282
 namelist, 235, 282–287
 forms, **208**
 Fortran 2003 compatibility, 3
 FORTRAN 77 compatibility, 4
 Fortran 90 compatibility, 4
 Fortran 95 compatibility, 3
 Fortran character set, **25**
 free source form, 30, **30**
 function, **13**, **510**
 function reference, **19**, 325
 function result, **510**
 FUNCTION statement, **329**
 function subprogram, **329**, **510**
function-name, 86, 301, 330, 331, 334, 336, 485, 488
function-reference (R1219), 78, 86, 133, **310**, 314, 325
function-stmt (R1228), 9, 300, 301, 329, 330, **330**, 331, 485, 488
function-subprogram (R1227), 9, 10, 290, **330**, 333

G

generic identifier, **303**, **510**
 generic interface, 71, **303**, **510**
 generic interface block, **301**, **510**
 generic name, **304**, **341**
 generic procedure reference, 306
 GENERIC statement, **70**
generic-name, 71, 300, 301, 486, 488
generic-spec (R1207), 70–72, 76, 101, 150, 162, 291, 292, 300, **300**, 301, 303, 486, 488
 global entity, **483**, **510**
 global identifier, **483**
 GO TO statement, **195**
goto-stmt (R851), 11, 38, 185, 195, **195**
 graphic character, **25**

H

hex-constant (R428), 56, **56**
hex-digit (R429), 56, **56**, 271
hex-digit-string (R1021), 271, **271**

host, **12**, **13**, **510**
 host association, **488**, **510**
 host instance, **165**
 host scoping unit, **12**, **510**

I

ICHAR intrinsic, 54
ID= specifier, 227, 250
 IEEE_, 352, 439–465
 IF construct, 190, **190**
 IF statement, 190, **191**
if-construct (R840), 11, 190, **190**
if-construct-name, 190
if-stmt (R845), 11, 38, 191, **191**
if-then-stmt (R841), 190, **190**, 195
imag-part (R419), 50, **50**
 image, **14**, **510**, **511**
 image control, **197**
 image index, **14**, **511**
 image selector, **125**
image-selector (R624), 20, 118, 119, **125**, 507
image-set (R862), 201, **201**, 202, 203
image-team (R860), 200, **200**, 203, 218, 221, 248, 254
 IMAGE_TEAM, **437**
 imaginary part, **50**
 implicit, **299**
 implicit interface, 310, **511**
 IMPLICIT statement, **107**
implicit-part (R205), 10, **10**
implicit-part-stmt (R206), 10, **10**
implicit-spec (R560), 108, **108**
implicit-stmt (R559), 10, **108**
 implied-shape array, **94**
implied-shape-spec (R518), 91, 94, **94**
 IMPORT statement, **302**
import-name, 302, 303
import-name-list, 303
import-stmt (R1209), 10, **302**
IMPURE, 329
 inactive, **186**
INCLUDE, 34
 INCLUDE line, **34**
index-name, 169–175, 186, 187, 486, 487, 500, 502
 inherit, **511**
 inheritance associated, **75**
 inheritance association, 497, **511**
 inherited, **75**
 initial point, **211**
initial-data-target (R448), 66, **66**, 67, 86, 87, 104
initial-proc-target (R1217), 67, 308, **308**, 309
 initialization, 67, 86, 87, 498, 613
initialization (R505), 86, **86**, 87
 initialization expression, **156**

initialization target, **67, 309**
initialization-expr (R730), 45, 66, 67, 86, 87, 94, 98, 106, 157, **157**
inner-shared-do-construct (R837), 185, **185**, 186
 input statement, **207**
input-item (R915), 223, 224, 229, **229**, 230, 243, 257, 503
 input/output editing, 259–287
 input/output list, 229
 input/output statements, 207–257
 INPUT_UNIT, 215, **437**
 inquire by file, **247**
 inquire by output list, **247**
 inquire by unit, **247**
 INQUIRE statement, **247**
inquire-spec (R930), 247, 248, **248**, 249, 257
inquire-stmt (R929), 11, **247**, 338
 inquiry function, **341, 511**
 inquiry, type parameter, 121
 instance, **333, 511**
int-constant (R308), 28, **28**, 103
int-constant-name, 48
int-constant-subobject (R541), 103, 104, **104**
int-expr (R727), 16, 44, 82, 118, 122, 125, 126, 139, 152, **152**, 154–157, 169, 180, 184, 186, 195, 201, 214, 215, 217, 224, 229, 232, 244, 248, 335
int-initialization-expr (R732), 47, 52, 61, 80, 81, 103, 157, **157**, 181, 196
int-literal-constant (R407), 28, 48, **48**, 52, 260–262
int-variable (R608), 117, **117**, 126, 217, 222, 224, 244–248, 251–256
int-variable-name, 184
INTEGER, 47
 integer constant, 28
 integer editing, 265
 integer model, 343
 integer type, **47, 47**
INTENT, 342
 intent, **511**
 INTENT attribute, **95**, 105
 INTENT statement, **105**
intent-spec (R523), 85, **95**, 105, 308
intent-stmt (R545), 11, **105**
 interface, **299**
 (procedure), 299
 abstract, **299**
 explicit, 299
 generic, 303
 implicit, 310
 interface block, **13, 300–303, 511**
 interface body, **13, 511**
 INTERFACE statement, **300**
interface-block (R1201), 10, **300**, 301
interface-body (R1205), 300, **300**, 301, 303, 488
interface-name (R1215), 308, **308**, 309
interface-name, 70–72
interface-specification (R1202), 300, **300**
interface-stmt (R1203), 300, **300**, 301, 303, 304, 488
 internal file, **214, 511**
 internal procedure, **13, 297, 511**
 internal subprogram, **12, 511**
 internal unit, **214**
internal-file-variable (R903), 214, **214**, 225, 503
internal-subprogram (R211), 10, **10**
internal-subprogram-part (R210), 9, **10**, 11, 289, 330–333
 interoperable, 472, **472, 475–477**, 478, **511**
 interoperates, **472**
 intrinsic, **22, 511**
 elemental, 341
 inquiry function, 341
 transformational, 341
 intrinsic assignment statement, **158**
 INTRINSIC attribute, **97**, 310
 intrinsic binary operation, **140**
 intrinsic module, **290**
 intrinsic operation, **140**
 intrinsic operations, 140–149
 intrinsic procedure, **297**, 341–435
 INTRINSIC statement, **310**
 intrinsic type, **17**, 47–55
 intrinsic unary operation, **140**
intrinsic-operator (R310), **28**, 29, 134, 137, 140, 150, 304
intrinsic-procedure-name, 310, 488
intrinsic-stmt (R1218), 11, **310**, 488
intrinsic-type-spec (R404), 45, **47**, 52
 invoke, **511**
io-control-spec (R913), 223, 224, **224**, 225, 238, 257
io-implied-do (R917), 229, **229**, 230, 231, 234, 257, 499, 501, 503, 544
io-implied-do-control (R919), 229, **229**, 232
io-implied-do-object (R918), 229, **229**
io-unit (R901), 214, **214**, 224, 225, 338
iomsg-variable (R907), 217, **218**, 222, 224, 244–246, 248, 255–257, 499
IOMSG= specifier, 257
IOSTAT= specifier, 256
 IOSTAT_END, **438**
 IOSTAT_INQUIRE_INTERNAL_UNIT, **438**
 ISO 10646 character set, **51**
 ISO 10646 character type, **51**, 158, 214, 219, 231, 264, 279, 410, 422
 ISO_C_BINDING module, 467
 ISO_FORTRAN_ENV module, 214, 215, 435
 iteration count, **40, 186**

K

k (R1013), 261, **261**, 268, 273, 276
 keyword, **21**, 313, **512**
keyword (R215), 21, **21**, 77, 78, 311
 kind, **47**, **48**, **50**, **51**, **55**
 KIND intrinsic, 47, 48, 50, 51, 55
 kind type parameter, **17**, 44, 47, 48, 50, 51, 55, **512**
kind-param (R408), 48, **48**, 49, 50, 52, 53, 55, 56
kind-selector (R405), 7, 47, **47**, 61

L

label, **512**
label (R313), 29, **29**, 184, 185, 195, 217, 218, 222–226, 244–246, 248, 249, 255, 256, 311
label-do-stmt (R824), 184, **184**, 185
language-binding-spec (R508), 85, **89**, 102, 330
lbracket (R469), 38, 63, 82, **82**, 86, 91, 102, 105, 107, 125, 126
 left tab limit, **275**
 length, **51**
 length of a character string, **512**
 length type parameter, 44
length-selector (R421), 7, 52, **52**
 letter, **25**
letter, 25, **25**, 27, 29, 108, 134, 136
letter-spec (R561), 108, **108**
level-1-expr (R702), 134, **134**, 138
level-2-expr (R706), 134, **134**, 135, 138
level-3-expr (R710), 135, **135**, 136
level-4-expr (R712), **135**, 136
level-5-expr (R717), 136, **136**, 137
 lexical token, **27**, **512**
 line, **30**, **512**
 linkage association, **512**
 list-directed formatting, 235, 278–282
 list-directed input/output statement, **226**
 literal constant, **18**, 118, **512**
literal-constant (R306), 28, **28**
 local entity, **512**
 local identifier, **483**, 484
 local variable, **18**, **512**
local-defined-operator (R1114), 291, 292, **292**
local-name, 291–293
LOGICAL, 55
 logical intrinsic assignment statement, **158**
 logical intrinsic operation, **140**, 146
 logical intrinsic operator, **140**
 logical type, 55, **55**
logical-expr (R724), 152, **152**, 157, 167, 180, 184, 186–188, 190, 191
logical-initialization-expr (R733), 157, **157**, 181
logical-literal-constant (R424), 28, **55**, 134, 137
logical-variable (R604), 117, **117**, 203
 loop, **184**

loop-control (R826), 184, **184**, 186, 187, 189
 low-level syntax, **27**
lower-bound (R513), 92, **92**, 93, 94
lower-bound-expr (R634), 126, **126**, 128, 164
lower-co-bound (R521), 95, **95**

M

m (R1008), 261, **261**, 265, 266, 271, 281
macro-actual-arg (R343), 37, 38, **38**
macro-actual-arg-value (R344), 38, **38**
macro-attr (R316), 35, **35**
macro-body-block (R322), 35, **35**, 36
macro-body-construct (R323), 35, **35**, 36
macro-body-stmt (R337), 35, 37, **37**
macro-condition (R333), 36, **36**
macro-declaration-stmt (R317), 35, **35**
macro-definition (R314), 10, 35, **35**, 36
macro-do-construct (R324), 35, **36**
macro-do-limit (R326), 36, **36**
macro-do-stmt (R325), 36, **36**, 40
macro-do-variable-name, 36, 40
macro-dummy-arg-name, 35
macro-dummy-arg-name-list, 35
macro-dummy-name, 38
macro-else-if-stmt (R330), 36, **36**
macro-else-stmt (R331), 36, **36**
macro-end-do-stmt (R327), 36, **36**
macro-end-if-stmt (R332), 36, **36**
macro-expr (R341), 35–37, **37**, 40
macro-if-construct (R328), 35, **36**
macro-if-then-stmt (R329), 36, **36**
macro-int-assignment-stmt (R334), 36, **36**
macro-integer-variable-name, 36
macro-local-variable-name, 35
macro-local-variable-name-list, 35
macro-name, 35, 37
macro-optional-decl-stmt (R319), 35, **35**
macro-tok-assignment-stmt (R335), 36, **36**
macro-tok-variable-name, 36
macro-type-declaration-stmt (R318), 35, **35**
macro-type-spec (R321), 35, **35**, 38
macro-variable-decl-stmt (R320), 35, **35**
 main program, 13, **289**, **512**
main-program (R1101), 9, 289, **289**
 many-one array section, **124**, **512**
mask-expr (R748), 166, 167, **167**, 168–173
 masked array assignment (WHERE), 166
masked-elsewhere-stmt (R749), 167, **167**, 168, 173
 model
 bit, 342
 integer, 343
 real, 343
 module, **14**, **290**, **512**
module (R1104), 9, **290**

module procedure, **13**, **298**, **512**
 module procedure interface, **301**, **512**
 module procedure interface body, **301**
 module reference, **22**
 MODULE statement, **290**
 module subprogram, **12**, **512**
module-name, 290–292, 488
module-nature (R1110), 291, **291**, 292
module-stmt (R1105), 9, 290, **290**
module-subprogram (R1108), 10, 290, **290**, 334
module-subprogram-part (R1107), 9, 10, 72, 76,
 290, **290**, 294, 559
mp-subprogram-stmt (R1238), 10, 333, **333**
mult-op (R708), 28, 134, **134**
mult-operand (R704), 134, **134**, 138

N

n (R1015), 261, 262, **262**
 name, **21**, **512**
name (R304), 6, 21, 27, **27**, 28, 86, 107, 117, 193,
 234, 308, 330
 name association, **487**, **512**
 name-value subsequences, **282**
NAME= specifier, 251
 named, **512**
 named common block, **113**
 named constant, **18**, 98, 106, 118, **513**
 named file, **208**
named-constant (R307), 28, **28**, 34, 50, 51, 80, 106,
 488
named-constant-def (R548), 106, **106**, 488
NAMED= specifier, 251
 namelist formatting, 235, 282–287
 namelist input/output statement, **226**
 NAMELIST statement, **110**
namelist-group-name, 110, 224–226, 233, 235, 259,
 282, 286, 293, 488, 503
namelist-group-object (R563), 110, **110**, 233, 234,
 236, 243, 257, 282, 283, 293
namelist-stmt (R562), 11, **110**, 488, 503
 Names, **27**
 NaN, 352, 444, **513**
nested-token-sequence (R347), 38, **38**
 next record, **211**
NEXTREC= specifier, 251
NML= specifier, 226
 nonadvancing input/output statement, **212**
nonblock-do-construct (R831), 184, **185**
 nonexecutable statement, **15**, **513**
 nonintrinsic module, **290**
nonlabel-do-stmt (R825), 184, **184**, 185
 normal, **444**
not-op (R718), 28, 136, **136**
 NOTIFY statement, **203**

notify-stmt (R863), 11, **203**
null-init (R506), 66, 67, 86, **86**, 87, 104, 308, 309
 NULLIFY statement, **130**
nullify-stmt (R638), 11, **130**, 503
NUMBER= specifier, 251
 numeric conversion, 160
 numeric editing, 265
 numeric intrinsic assignment statement, **158**
 numeric intrinsic operation, **140**, 142
 numeric intrinsic operator, **140**
 numeric relational intrinsic operation, **140**
 numeric sequence type, **59**, 111, 496, 618
 numeric storage unit, **493**, **513**
 numeric type, **47**, **513**
numeric-expr (R728), 49, 152, **152**, 195
 NUMERIC_STORAGE_SIZE, **438**

O

object, *see* data object, **18**, **513**
 object designator, **21**, **513**
object-name (R504), 86, **86**, 102, 106, 107, 117, 488
 obsolescent feature, **7**, **513**
octal-constant (R427), 56, **56**
only (R1112), 291, **291**, 292
only-use-name (R1113), 291, 292, **292**
 OPEN statement, **217**
open-stmt (R904), 12, **217**, 338
OPENED= specifier, 251
 operand, **23**, **513**
 operation, **513**
 defined, **150**
 extension, 151
 intrinsic, **140**
 intrinsic bits, 147
 logical intrinsic, 146
 numeric intrinsic, 142
 relational intrinsic, 148
 operations, 43
 operator, **23**, 28, **513**
 operator precedence, 137
 OPTIONAL attribute, **97**, 106
 optional dummy argument, 322
 OPTIONAL statement, **106**
optional-stmt (R546), 11, **106**
or-op (R720), 28, 136, **136**
or-operand (R715), 136, **136**
outer-shared-do-construct (R835), 185, **185**
 output statement, **207**
output-item (R916), 223, 224, 229, **229**, 243, 247
 OUTPUT_UNIT, 215, **438**
 override, 76, **513**
 overrides, **67**

P

PAD= specifier, 220, 228, 251

PARAMETER attribute, **98**, 106

PARAMETER statement, **106**

parameter-stmt (R547), 10, **106**, 488

parent, **294**

parent component, **75**, **513**

parent data transfer statement, **236**, 236–240, 257

parent type, **74**, **513**

parent-identifier (R1118), 294, **294**

parent-string (R610), 90, 118, **118**

parent-submodule-name, 294

parent-type-name, 58

parentheses, 152

part-name, 118, 119, 122

part-ref (R613), 90, 103, 111, 118, **118**, 119, 122, 123, 319

partially [storage] associated, **495**

passed-object dummy argument, **66**, 314, **513**

PENDING= specifier, 251

pointer, **20**, **513**

pointer assignment, 163, **513**

pointer assignment statement, **513**

pointer associated, **513**

pointer association, 491, **513**

pointer association context, 503

pointer association status, **491**

POINTER attribute, **98**, 106

POINTER statement, **106**

pointer-assignment-stmt (R735), 12, 99, **163**, 169, 172, 338, 503

pointer-decl (R550), 106, **106**

pointer-object (R639), 130, **130**, 503

pointer-stmt (R549), 11, **106**, 488

polymorphic, **46**, **513**

POS= specifier, 228, 252

position, **209**

position-edit-desc (R1014), 261, **261**

position-spec (R926), 244, 245, **245**

POSITION= specifier, 220, 252

positional arguments, 341

power-op (R707), 28, 134, **134**

pre-existing, **497**

precedence of operators, 137

preceding record, **211**

preconnected, **514**

preconnected files, 216

Preconnection, **216**

prefix (R1225), 329, **329**, 330, 332

prefix-spec (R1226), 329, **329**, 337, 339

present, **321**

present (dummy argument), 321

PRESENT intrinsic, 98

primaries, 336

primary, 133

primary (R701), 133, **133**, 134

PRINT statement, **223**

print-stmt (R912), 12, **223**, 338

PRIVATE attribute, 58, **88**

PRIVATE statement, 69, 70, **101**

private-components-stmt (R449), 58, 69, **69**

private-or-sequence (R433), 57, 58, **58**

proc-attr-spec (R1213), 308, **308**, 309

proc-component-attr-spec (R446), 64, **64**

proc-component-def-stmt (R445), 62, 64, **64**

proc-component-ref (R741), 164, **164**, 311

proc-decl (R1214), 64, 67, 308, **308**, 309

proc-entity-name, 106, 309

proc-interface (R1212), 64, 308, **308**, 309

proc-language-binding-spec (R1229), 88, 308, 309, 329, 330, **330**, 331, 332, 336, 477

proc-pointer-init (R1216), 308, **308**

proc-pointer-name (R554), 107, **107**, 113, 130, 164, 488

proc-pointer-object (R740), 163, **164**, 165, 172, 356, 503

proc-target (R742), 77–79, 99, 163, 164, **164**, 165, 172, 322, 356, 493

procedure, **13**, **514**

characteristics of, 298

dummy, 298

elemental, 339

external, 297

internal, 297

intrinsic, 341–435

non-Fortran, 336

pure, 337

type-bound, **71**, 328

procedure declaration statement, **308**

procedure designator, **21**, **514**

procedure interface, 299, **514**

procedure pointer, **20**, 308

procedure reference, **22**, 310

generic, 306

resolving, 326

type-bound, 328

PROCEDURE statement, **303**

procedure-component-name, 164

procedure-declaration-stmt (R1211), 10, 308, **308**, 309, 488

procedure-designator (R1221), 310, 311, **311**, 328

procedure-entity-name, 308

procedure-name, 70, 71, 164, 165, 300, 301, 308, 311, 333

procedure-stmt (R1206), 300, **300**, 301

processor, **1**, **514**

processor dependent, **3**, **514**

program, **12**, **514**

program (R201), 9, **9**
 PROGRAM statement, **289**
 program unit, **12**, **514**
program-name, 289
program-stmt (R1102), 9, 289, **289**
program-unit (R202), 7, 9, **9**
 PROTECTED attribute, **98**, 106
 PROTECTED statement, **106**
protected-stmt (R551), 11, **106**
 prototype, **514**
 PUBLIC attribute, 58, **88**
 PUBLIC statement, **101**
PURE, 329
 pure procedure, **337**, **514**

Q

QUERY statement, **203**
query-spec (R865), 203, **203**
query-stmt (R864), 12, **203**

R

r (R1005), 260, **260**, 261, 263
 range, **186**
 RANGE intrinsic, 47
 rank, **19**, **514**
rbracket (R470), 38, 63, 82, **82**, 86, 91, 102, 105, 107, 125, 126
 READ statement, **223**
read-stmt (R910), 12, **223**, 224, 338, 503
READ= specifier, 252
 reading, **207**
READWRITE= specifier, 253
REAL, 49
 real and complex editing, 266
 real model, 343
 real part, **50**
 real type, **48**, 48–50
real-literal-constant (R413), 28, 49, **49**
real-part (R418), 50, **50**
REC= specifier, 228
RECL= specifier, 220, 253
 record, **207**, **514**
 record file, **207**
 record length, **208**
 record number, **210**
RECURSIVE, 329
 recursive input/output statement, **257**
 reference, **291**, **514**
rel-op (R713), 28, 135, **135**, 149
 relational intrinsic operation, **140**, 148
 relational intrinsic operator, **140**
rename (R1111), 291, **291**, 292, 484
rep-char, 53, **53**, 262, 279, 284
 repeat specification., **260**

representable character, **53**
 representation method, **47**, **48**, **51**, **55**
 resolving procedure reference, 326
 resolving procedure references
 derived-type input/output, 242
 restricted expression, **154**
 result variable, **13**, **514**
result-name, 330, 331, 334, 488
result-token (R338), 37, **37**
 RETURN statement, **335**
return-stmt (R1241), 12, 16, 38, 185, 335, **335**
 REWIND statement, **246**
rewind-stmt (R925), 12, **244**, 338
round-edit-desc (R1018), 261, **262**
ROUND= specifier, 221, 228, 253
 rounding mode, **514**

S

satisfied, **203**
 SAVE attribute, 103, 107
 SAVE statement, **107**
save-stmt (R552), 11, **107**, 488
 saved, **100**
saved-entity (R553), 107, **107**, 180
 scalar, **19**, **514**
scalar-xyz (R103), 6, **6**
 scale factor, **261**
 scope, **483**, **514**
 scoping unit, **12**, **514**
 section subscript, **515**
section-subscript (R620), 118, 119, 122, **122**, 123, 319
 segment, **198**
 SELECT CASE statement, 180
 SELECT TYPE construct, 192, **192**, 487
 SELECT TYPE statement, **192**, 490
select-case-stmt (R811), 180, **180**, 195
select-construct-name, 192
select-type-construct (R846), 11, 192, **192**
select-type-stmt (R847), 192, **192**, 195
 SELECTED_INT_KIND intrinsic, 47
 selector, **515**
selector (R805), 178, **178**, 179, 192, 193, 322, 491, 503
 separate module procedure, **333**
 separate module subprogram statement, **333**
separate-module-subprogram (R1237), 10, 290, 333, **333**
 sequence, **23**
 sequence association, 321
 SEQUENCE property, 60
 SEQUENCE statement, **59**
 sequence structure, **57**
 sequence type, **59**

sequence-stmt (R435), 58, **59**
 sequential access, 209
 sequential access input/output statement, **228**
SEQUENTIAL= specifier, 253
 shape, **19, 515**
 shape conformance, **154**
shared-term-do-construct (R836), 185, **185**
sign (R411), 48, **48**, 49, 266
sign-edit-desc (R1016), 261, **262**
SIGN= specifier, 221, 228, 253
signed-digit-string (R409), **48**, 50, 266
signed-int-literal-constant (R406), 48, **48**, 50, 104, 261
signed-real-literal-constant (R412), **49**, 50, 51, 104
significand (R414), 49, **49**
 size, **19, 515**
 size of a common block, **114**
 size of a storage sequence, **493**
SIZE= specifier, 228, 253
 source forms, **30**
source-expr (R630), 126, **126**, 127, 128
 special characters, **26**
special-character, 25, **27**
 specific interface, **301**
 specific interface block, **301**
 specific name, **341**
 specification, 85–116
 specification expression, **154, 515**
 specification function, **155, 515**
 specification inquiry, **155**
specification-expr (R729), 45, 52, 86, 92, 95, 154, **154**, 339
specification-part (R204), 9, 10, **10**, 15, 16, 70, 88, 101, 155–157, 179, 180, 289, 290, 294, 295, 300, 330, 332, 333, 337
specification-stmt (R212), 10, **11**
 standard-conforming program, **2, 515**
 starting point, **118**
stat-variable (R628), 126, **126**, 127, 130, 132, 199, 503, 594
STAT= specifier, 132
 statement, **30, 515**

- accessibility, 101
- ALLOCATABLE, 102
- ALLOCATE, 125
- arithmetic IF, 195
- assignment, 157
- ASSOCIATE, 178, 490
- ASYNCHRONOUS, 102
- attribute specification, 101–116
- BACKSPACE, 245
- BIND, 102
- BLOCK, 179
- BLOCK DATA, 294
- CALL, 311
- CASE, 180
- CLASS DEFAULT, 192
- CLASS IS, 192
- CLOSE, 222
- COMMON, 113–116
- component definition, 62
- computed GO TO, 195
- CONTAINS, 70, 336
- CONTIGUOUS, 102
- CONTINUE, 196
- CRITICAL, 183
- CYCLE, 187
- DATA, 102
- data transfer, 223
- DEALLOCATE, 130
- DIMENSION, 105
- DO, 184
- DO CONCURRENT, 184
- DO WHILE, 184
- END, 16
- END BLOCK, 179
- END CRITICAL, 183
- END INTERFACE, 300
- ENDFILE, 245
- ENTRY, 334
- EQUIVALENCE, 110–113
- executable, 15
- EXIT, 194
- EXTERNAL, 307
- file inquiry, 247
- file positioning, 244
- FINAL, 72
- FLUSH, 246
- FORALL, 174
- FORMAT, 259
- formatted input/output, 225
- FUNCTION, 329
- GENERIC, 70
- GO TO, 195
- IF, 191
- IMPLICIT, 107
- IMPORT, 302
- input/output, 207–257
- INTENT, 105
- INTERFACE, 300
- INTRINSIC, 310
- list-directed input/output, 226
- MODULE, 290
- NAMELIST, 110
- namelist input/output, 226
- NOTIFY, 203
- NULLIFY, 130
- OPEN, 217

- OPTIONAL, 106
- PARAMETER, 106
- POINTER, 106
- pointer assignment, 163
- PRINT, 223
- PRIVATE, 69, 70, 101
- PROCEDURE, 303
- procedure declaration, 308
- PROGRAM, 289
- PROTECTED, 106
- PUBLIC, 101
- QUERY, 203
- READ, 223
- RETURN, 335
- REWIND, 246
- SAVE, 107
- SELECT CASE, 180
- SELECT TYPE, 192, 490
- separate module subprogram, 333
- SEQUENCE, 59
- statement function, 336
- STOP, 196
- SUBMODULE, 294
- SUBROUTINE, 332
- SYNC ALL, 199
- SYNC IMAGES, 201
- SYNC MEMORY, 204
- SYNC TEAM, 200
- TARGET, 107
- TYPE, 57
- type declaration, 85–87
- TYPE IS, 192
- type parameter definition, 61
- type-bound procedure, 70
- unformatted input/output, 225
- USE, 291
- VALUE, 107
- VOLATILE, 107
- WAIT, 244
- WHERE, 166
- WRITE, 223
- statement entity, **483, 515**
- statement function, **298, 336, 515**
- statement function statement, **336**
- statement label, 29, **29, 515**
- statement order, 15
- statements
 - INQUIRE, 247
- STATUS= specifier**, 221, 223
- stmt-function-stmt* (R1243), 10, 290, 294, 301, **336**, 488
- STOP statement, **196**
- stop-code* (R856), 196, **196**, 197
- stop-stmt* (R855), 12, 38, 185, **196**, 338
- storage associated, **495**
- storage association, 110–116, 493, **493, 515**
- storage sequence, 114, **493, 515**
- storage unit, **493, 515**
- stream access, 210
- stream access input/output statement, **228**
- stream file, **207**
- STREAM= specifier**, 253
- stride, **124, 515**
- stride* (R622), 122, **122**, 169, 170, 173, 232, 319
- string, *see* character string
- struct, **515**
- structure, **17, 57, 515**
- structure component, **118, 515**
- structure constructor, **77, 516**
- structure-component* (R614), 103, 117, 118, **119**, 126, 130
- structure-constructor* (R459), **77**, 78, 104, 133, 338
- subcomponent, **120, 516**
- submodule, **14, 294, 516**
- submodule* (R1116), 9, **294**
- submodule identifier, **294**
- SUBMODULE statement, **294**
- submodule-name*, 294
- submodule-stmt* (R1117), 9, 294, **294**
- subobject, **18, 516**
- subprogram, **12, 516**
- subroutine, **13, 516**
- subroutine reference, 325
- SUBROUTINE statement, **332**
- subroutine subprogram, **332, 516**
- subroutine-name*, 301, 332, 333, 485
- subroutine-stmt* (R1234), 9, 300, 301, 329, 330, 332, **332**, 333, 485, 488
- subroutine-subprogram* (R1233), 9, 10, 290, **332**, 333
- subscript, **516**
- subscript* (R619), 7, 103, 122, **122**, 169, 170, 173, 232, 319
- subscript triplet, 124, **516**
- subscript-triplet* (R621), 122, **122**, 123, 319
- substring, **118, 516**
- substring* (R609), 111, 117, **118**
- substring-range* (R611), 90, 118, **118**, 120, 122, 232, 319
- suffix* (R1231), 330, **331**, 334
- SYNC ALL statement, **199**
- SYNC IMAGES statement, **201**
- SYNC MEMORY statement, **204**
- SYNC TEAM statement, **200**
- sync-all-stmt* (R857), 12, **199**
- sync-images-stmt* (R861), 12, **201**
- sync-memory-stmt* (R866), 12, **204**
- sync-stat* (R858), 199, **199**, 200, 201, 203, 204

sync-team-stmt (R859), 12, **200**

synchronous, **226**, 228, 231

T

target, **20**, **516**

TARGET attribute, **100**, 107, 315

TARGET statement, **107**

target-decl (R556), 107, **107**

target-stmt (R555), 11, **107**, 488

team, **14**, **516**

team synchronization, **200**, 223, 341, 382, **516**

TEAM= specifier, 221, 254

terminal point, **211**

TKR compatible, **306**

token (R339), 37, **37**

totally [storage] associated, **495**

transfer of control, 177, 195, 255, 256

transformational function, **341**, **516**

type, **17**, 43–82, **516**

abstract, **74**

base, 74

bits, 55–57

character, 51–55

complex, 50

concept, **43**

declared, 46

derived, 57–80

dynamic, 46

expression, 152

extended, 74

extensible, 74

extension, 74

integer, 47

intrinsic, 47–55

logical, 55

operation, 153

parent, 74

primary, 152

real, 48–50

type compatible, **46**, **516**

type conformance, 158

type declaration statement, 85–87, **516**

type equality, 60

type incompatible, **46**

TYPE IS statement, **192**

type parameter, **17**, 44, 47, 48, 485, **516**

type parameter definition statement, **61**

type parameter inquiry, **121**

type parameter keyword, **21**

type parameter order, **62**, **517**

type specifier, **45**

derived type, 46

TYPE, 46

TYPE statement, **57**

TYPE type specifier, 46

type-attr-spec (R432), 57, **57**, 58

type-bound procedure, **71**, 328, **516**

Type-bound procedure statement, **70**

type-bound-generic-stmt (R454), 70, **70**

type-bound-proc-binding (R452), 70, **70**

type-bound-procedure-part (R450), 57, 59, **70**, 72, 475

type-bound-procedure-stmt (R453), 70, **70**

type-declaration-stmt (R501), 10, 52, 85, **85**, 337, 488

type-guard-stmt (R848), 192, **192**

type-name, 57, 58, 61, 71, 77, 516

type-param-attr-spec (R438), 61, **61**

type-param-decl (R437), 61, **61**

type-param-def-stmt (R436), 57, 61, **61**

type-param-inquiry (R616), 121, **121**, 133, 486

type-param-name, 57, 61, 64, 121, 133, 134, 486, 488

type-param-spec (R458), 77, **77**

type-param-value (R401), 44, **44**, 45, 52, 64, 77, 126, 127

type-spec (R402), **45**, 46, 52, 53, 82, 125–127, 169, 170, 192, 487

U

ultimate component, **57**, **517**

unallocated, **129**

undefined, **22**, **517**

undefinition of variables, 497

underscore (R303), 25, **25**

unformatted data transfer, 234

unformatted input/output statement, **225**

unformatted record, **208**

UNFORMATTED= specifier, 254

Unicode, **219**

unit, **214**

unlimited polymorphic, **46**

unlimited-format-item (R1004), 259, **260**, 263

unsaved, **100**

unsigned, **517**

unspecified storage unit, **494**, **517**

upper-bound (R514), 92, **92**

upper-bound-expr (R635), 126, **126**, 128, 164

upper-co-bound (R522), 95, **95**, 369

use associated, **291**

use association, 488, **517**

USE statement, **291**

use-defined-operator (R1115), 291, 292, **292**, 293

use-name, 101, 291–293, 484

use-stmt (R1109), 10, **291**, 292, 488

V

v (R1011), 239, 261, **261**, 274

VALUE attribute, **100**, 107, 314

value separator, **278**

VALUE statement, **107**

value-stmt (R557), 11, **107**

variable, **18**, **517**

variable (R601), xv, 78, 103, 117, **117**, 158–161, 163, 164, 168, 170, 172, 178, 192, 193, 229, 311, 462, 503

variable-name (R602), 110, 111, 113, 114, 117, **117**, 118, 126, 130, 163, 488, 503

variables

definition & undefinition, 497

vector subscript, **124**, **517**

vector-subscript (R623), 122, **122**, 123

void, **517**

VOLATILE attribute, **100**, 107

VOLATILE statement, **107**

volatile-stmt (R558), 11, **107**

W

w (R1007), 261, **261**, 265–269, 271–274, 279, 281, 284

wait operation, 222, 231, 243, **243**

WAIT statement, **244**

wait-spec (R922), 244, **244**

wait-stmt (R921), 12, **244**, 338

WHERE construct, 166

WHERE statement, 166

where-assignment-stmt (R747), 166, 167, **167**, 168, 169, 173, 508

where-body-construct (R746), 166, 167, **167**, 168

where-construct (R744), 11, **166**, 167, 169, 173

where-construct-name, 167

where-construct-stmt (R745), 166, 167, **167**, 173, 195

where-stmt (R743), 12, **166**, 167, 169, 173

whole array, **121**, **517**

WRITE statement, **223**

write-stmt (R911), 12, **223**, 224, 338, 503

WRITE= specifier, 254

writing, **207**

X

xyz-list (R101), **6**

xyz-name (R102), **6**

Z

zero-size array, 19, 92, 104

ZZZUTI005, 196

ZZZUTI006, 197

ZZZUTI010, 290

ZZZUTI016, 314

ZZZUTI050, 315

ZZZUTI073, 437

ZZZUTI076, 500

ZZZUTI080, xv

ZZZUTI081, 128

ZZZUTI082, 567

ZZZUTI084, 180

ZZZUTI086, 200

ZZZUTI089, 120

ZZZUTI090, 437

ZZZUTI091, 232

ZZZUTI092, 420

ZZZUTI093, 424

ZZZUTI094, 63

ZZZUTI096, 318

ZZZUTI097, 317

ZZZUTI098, 440

ZZZUTI099, 2

ZZZUTI100, 596

ZZZUTI101, 320

ZZZUTI103, 330

ZZZUTI104, 142

ZZZUTI105, 368

ZZZUTI106, 247

ZZZUTI107, 596

ZZZUTI108, 597

ZZZUTI192, 502

ZZZUTI5003, 512