

**WORKING DRAFT**

**J3/06-007**

**July 11, 2006 18:25**

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 . . . . .	6
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 . . . . .	7
2	Fortran terms and concepts . . . . .	9
2.1	High level syntax . . . . .	9
2.2	Program unit concepts . . . . .	12
2.2.1	Program . . . . .	12
2.2.2	Main program . . . . .	12
2.2.3	Procedure . . . . .	12
2.2.4	Module . . . . .	13
2.2.5	Submodule . . . . .	13
2.3	Execution concepts . . . . .	14
2.3.1	Executable/nonexecutable statements . . . . .	14
2.3.2	Statement order . . . . .	14
2.3.3	The END statement . . . . .	15
2.3.4	Execution sequence . . . . .	15
2.3.5	Images . . . . .	17
2.4	Data concepts . . . . .	18
2.4.1	Type . . . . .	18
2.4.2	Data value . . . . .	19
2.4.3	Data entity . . . . .	19
2.4.4	Scalar . . . . .	20
2.4.5	Array . . . . .	20

2.4.6	Co-array . . . . .	21
2.4.7	Pointer . . . . .	22
2.4.8	Storage . . . . .	22
2.5	Fundamental terms . . . . .	22
2.5.1	Name and designator . . . . .	22
2.5.2	Keyword . . . . .	22
2.5.3	Association . . . . .	23
2.5.4	Declaration . . . . .	23
2.5.5	Definition . . . . .	23
2.5.6	Reference . . . . .	23
2.5.7	Intrinsic . . . . .	23
2.5.8	Operator . . . . .	24
2.5.9	Sequence . . . . .	24
2.5.10	Companion processors . . . . .	24
3	Characters, lexical tokens, and source form . . . . .	25
3.1	Processor character set . . . . .	25
3.1.1	Letters . . . . .	25
3.1.2	Digits . . . . .	25
3.1.3	Underscore . . . . .	26
3.1.4	Special characters . . . . .	26
3.1.5	Other characters . . . . .	26
3.2	Low-level syntax . . . . .	26
3.2.1	Names . . . . .	27
3.2.2	Constants . . . . .	27
3.2.3	Operators . . . . .	27
3.2.4	Statement labels . . . . .	28
3.2.5	Delimiters . . . . .	29
3.3	Source form . . . . .	29
3.3.1	Free source form . . . . .	29
3.3.2	Fixed source form . . . . .	31
3.4	Including source text . . . . .	32
3.5	Macro processing . . . . .	33
3.5.1	Macro definition . . . . .	33
3.5.2	Macro expansion . . . . .	35
4	Types . . . . .	41
4.1	The concept of type . . . . .	41
4.1.1	Set of values . . . . .	41
4.1.2	Constants . . . . .	41
4.1.3	Operations . . . . .	41
4.2	Type parameters . . . . .	41
4.3	Relationship of types and values to objects . . . . .	43
4.3.1	Type specifiers and type compatibility . . . . .	43
4.4	Intrinsic types . . . . .	46
4.4.1	Classification and specification . . . . .	46
4.4.2	Integer type . . . . .	46
4.4.3	Real type . . . . .	47
4.4.4	Complex type . . . . .	49
4.4.5	Character type . . . . .	50
4.4.6	Logical type . . . . .	54
4.4.7	Bits type . . . . .	54
4.5	Derived types . . . . .	56
4.5.1	Derived type concepts . . . . .	56

4.5.2	Derived-type definition . . . . .	56
4.5.3	Derived-type parameters . . . . .	60
4.5.4	Components . . . . .	61
4.5.5	Type-bound procedures . . . . .	68
4.5.6	Final subroutines . . . . .	71
4.5.7	Type extension . . . . .	73
4.5.8	Derived-type values . . . . .	75
4.5.9	Derived-type specifier . . . . .	75
4.5.10	Construction of derived-type values . . . . .	76
4.5.11	Derived-type operations and assignment . . . . .	78
4.6	Enumerations and enumerators . . . . .	78
4.7	Construction of array values . . . . .	80
5	Attribute declarations and specifications . . . . .	83
5.1	General . . . . .	83
5.2	Type declaration statements . . . . .	83
5.2.1	Syntax . . . . .	83
5.2.2	Initialization . . . . .	85
5.2.3	Examples of type declaration statements . . . . .	85
5.3	Attributes . . . . .	85
5.3.1	Constraints . . . . .	85
5.3.2	Accessibility attribute . . . . .	86
5.3.3	ALLOCATABLE attribute . . . . .	86
5.3.4	ASYNCHRONOUS attribute . . . . .	86
5.3.5	BIND attribute for data entities . . . . .	87
5.3.6	CONTIGUOUS attribute . . . . .	87
5.3.7	DIMENSION attribute . . . . .	88
5.3.8	EXTERNAL attribute . . . . .	92
5.3.9	INTENT attribute . . . . .	92
5.3.10	INTRINSIC attribute . . . . .	94
5.3.11	OPTIONAL attribute . . . . .	94
5.3.12	PARAMETER attribute . . . . .	95
5.3.13	POINTER attribute . . . . .	95
5.3.14	PROTECTED attribute . . . . .	96
5.3.15	SAVE attribute . . . . .	96
5.3.16	TARGET attribute . . . . .	98
5.3.17	VALUE attribute . . . . .	99
5.3.18	VOLATILE attribute . . . . .	99
5.4	Attribute specification statements . . . . .	99
5.4.1	Accessibility statements . . . . .	99
5.4.2	ALLOCATABLE statement . . . . .	100
5.4.3	ASYNCHRONOUS statement . . . . .	100
5.4.4	BIND statement . . . . .	100
5.4.5	CONTIGUOUS statement . . . . .	101
5.4.6	DATA statement . . . . .	101
5.4.7	DIMENSION statement . . . . .	103
5.4.8	INTENT statement . . . . .	103
5.4.9	OPTIONAL statement . . . . .	104
5.4.10	PARAMETER statement . . . . .	104
5.4.11	POINTER statement . . . . .	104
5.4.12	PROTECTED statement . . . . .	105
5.4.13	SAVE statement . . . . .	105
5.4.14	TARGET statement . . . . .	105
5.4.15	VALUE statement . . . . .	105

5.4.16	VOLATILE statement . . . . .	105
5.5	IMPLICIT statement . . . . .	106
5.6	NAMelist statement . . . . .	108
5.7	Storage association of data objects . . . . .	108
5.7.1	EQUIVALENCE statement . . . . .	108
5.7.2	COMMON statement . . . . .	111
5.7.3	Restrictions on common and equivalence . . . . .	114
6	Use of data objects . . . . .	115
6.1	Scalars . . . . .	116
6.1.1	Substrings . . . . .	116
6.1.2	Structure components . . . . .	117
6.1.3	Complex parts . . . . .	118
6.1.4	Type parameter inquiry . . . . .	119
6.2	Arrays . . . . .	119
6.2.1	Whole arrays . . . . .	119
6.2.2	Array elements and array sections . . . . .	120
6.2.3	Image selectors . . . . .	123
6.3	Dynamic association . . . . .	123
6.3.1	ALLOCATE statement . . . . .	124
6.3.2	NULLIFY statement . . . . .	128
6.3.3	DEALLOCATE statement . . . . .	128
7	Expressions and assignment . . . . .	133
7.1	Expressions . . . . .	133
7.1.1	Form of an expression . . . . .	133
7.1.2	Intrinsic operations . . . . .	137
7.1.3	Defined operations . . . . .	138
7.1.4	Type, type parameters, and shape of an expression . . . . .	139
7.1.5	Conformability rules for elemental operations . . . . .	141
7.1.6	Specification expression . . . . .	142
7.1.7	Initialization expression . . . . .	143
7.1.8	Evaluation of operations . . . . .	145
7.2	Interpretation of operations . . . . .	150
7.2.1	General . . . . .	150
7.2.2	Numeric intrinsic operations . . . . .	150
7.2.3	Character intrinsic operation . . . . .	151
7.2.4	Relational intrinsic operations . . . . .	152
7.2.5	Logical intrinsic operations . . . . .	153
7.2.6	Bits intrinsic operations . . . . .	154
7.3	Precedence of operators . . . . .	154
7.4	Assignment . . . . .	156
7.4.1	Assignment statement . . . . .	156
7.4.2	Pointer assignment . . . . .	162
7.4.3	Masked array assignment – WHERE . . . . .	166
7.4.4	FORALL . . . . .	168
8	Execution control . . . . .	175
8.1	Executable constructs containing blocks . . . . .	175
8.1.1	General . . . . .	175
8.1.2	Rules governing blocks . . . . .	176
8.1.3	IF construct . . . . .	176
8.1.4	CASE construct . . . . .	178
8.1.5	ASSOCIATE construct . . . . .	180

8.1.6	SELECT TYPE construct . . . . .	182
8.1.7	DO construct . . . . .	185
8.1.8	EXIT statement . . . . .	191
8.1.9	BLOCK construct . . . . .	191
8.1.10	CRITICAL construct . . . . .	192
8.2	Branching . . . . .	193
8.2.1	GO TO statement . . . . .	193
8.2.2	Computed GO TO statement . . . . .	193
8.2.3	Arithmetic IF statement . . . . .	193
8.3	CONTINUE statement . . . . .	194
8.4	STOP statement . . . . .	194
8.5	Image execution control . . . . .	196
8.5.1	Image control statements . . . . .	196
8.5.2	SYNC_ALL statement . . . . .	198
8.5.3	SYNC_TEAM statement . . . . .	199
8.5.4	SYNC_IMAGES statement . . . . .	201
8.5.5	NOTIFY and QUERY statements . . . . .	202
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 . . . . .	217
9.4.5	OPEN statement . . . . .	217
9.4.6	NEWUNIT= specifier in the OPEN statement . . . . .	220
9.4.7	CLOSE statement . . . . .	223
9.5	Data transfer statements . . . . .	224
9.5.1	Control information list . . . . .	224
9.5.2	Data transfer input/output list . . . . .	229
9.5.3	Execution of a data transfer input/output statement . . . . .	232
9.5.4	Termination of data transfer statements . . . . .	243
9.6	Waiting on pending data transfer . . . . .	244
9.6.1	WAIT statement . . . . .	244
9.6.2	Wait operation . . . . .	244
9.7	File positioning statements . . . . .	245
9.7.1	BACKSPACE statement . . . . .	246
9.7.2	ENDFILE statement . . . . .	246
9.7.3	REWIND statement . . . . .	246
9.8	FLUSH statement . . . . .	247
9.9	File inquiry . . . . .	248
9.9.1	Inquiry specifiers . . . . .	248

9.9.2	Restrictions on inquiry specifiers . . . . .	254
9.9.3	Inquire by output list . . . . .	254
9.10	Error, end-of-record, and end-of-file conditions . . . . .	255
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 . . . . .	256
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 . . . . .	272
10.7.4	Character editing . . . . .	272
10.7.5	Generalized editing . . . . .	273
10.7.6	User-defined derived-type editing . . . . .	274
10.8	Control edit descriptors . . . . .	275
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 . . . . .	277
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 . . . . .	278
10.9	Character string edit descriptors . . . . .	278
10.10	List-directed formatting . . . . .	278
10.10.1	List-directed input . . . . .	279
10.10.2	List-directed output . . . . .	281
10.11	Namelist formatting . . . . .	283
10.11.1	Namelist input . . . . .	283
10.11.2	Namelist output . . . . .	287
11	Program units . . . . .	289
11.1	Main program . . . . .	289
11.2	Modules . . . . .	290
11.2.1	The USE statement and use association . . . . .	291
11.2.2	Submodules . . . . .	293
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 of procedures . . . . .	298
12.3.1	Characteristics of dummy arguments . . . . .	298
12.3.2	Characteristics of function results . . . . .	299
12.4	Procedure interface . . . . .	299
12.4.1	Implicit and explicit interfaces . . . . .	299
12.4.2	Specification of the procedure interface . . . . .	300
12.5	Procedure reference . . . . .	309
12.5.1	Actual arguments, dummy arguments, and argument association . . . . .	311
12.5.2	Function reference . . . . .	322
12.5.3	Subroutine reference . . . . .	322
12.5.4	Resolving named procedure references . . . . .	323
12.5.5	Resolving type-bound procedure references . . . . .	325
12.6	Procedure definition . . . . .	325
12.6.1	Intrinsic procedure definition . . . . .	325
12.6.2	Procedures defined by subprograms . . . . .	325
12.6.3	Definition and invocation of procedures by means other than Fortran . . . . .	332
12.6.4	Statement function . . . . .	333
12.7	Pure procedures . . . . .	333
12.8	Elemental procedures . . . . .	335
12.8.1	Elemental procedure declaration and interface . . . . .	335
12.8.2	Elemental function actual arguments and results . . . . .	336
12.8.3	Elemental subroutine actual arguments . . . . .	337
13	Intrinsic procedures and modules . . . . .	339
13.1	Classes of intrinsic procedures . . . . .	339
13.2	Arguments to intrinsic procedures . . . . .	340
13.2.1	General rules . . . . .	340
13.2.2	The shape of array arguments . . . . .	341
13.2.3	Mask arguments . . . . .	341
13.2.4	Arguments of collective subroutines . . . . .	341
13.3	Bit model . . . . .	341
13.4	Numeric models . . . . .	342
13.5	Standard generic intrinsic procedures . . . . .	343
13.5.1	Numeric functions . . . . .	343
13.5.2	Mathematical functions . . . . .	343
13.5.3	Character functions . . . . .	344
13.5.4	Kind functions . . . . .	345
13.5.5	Miscellaneous type conversion functions . . . . .	345
13.5.6	Numeric inquiry functions . . . . .	345
13.5.7	Array inquiry functions . . . . .	345
13.5.8	Other inquiry functions . . . . .	345
13.5.9	Bit manipulation procedures . . . . .	346
13.5.10	Floating-point manipulation functions . . . . .	346
13.5.11	Vector and matrix multiply functions . . . . .	346
13.5.12	Array reduction functions . . . . .	346
13.5.13	Array construction functions . . . . .	347
13.5.14	Array location functions . . . . .	347
13.5.15	Collective subroutines . . . . .	347
13.5.16	Null function . . . . .	348
13.5.17	Allocation transfer procedure . . . . .	348

13.5.18	Random number subroutines . . . . .	348
13.5.19	System environment procedures . . . . .	348
13.6	Specific names for standard intrinsic functions . . . . .	349
13.7	Specifications of the standard intrinsic procedures . . . . .	350
13.8	Standard intrinsic modules . . . . .	437
13.8.1	General . . . . .	437
13.8.2	The IEEE modules . . . . .	437
13.8.3	The ISO_FORTRAN_ENV intrinsic module . . . . .	437
13.8.4	The ISO_C_BINDING module . . . . .	441
14	Exceptions and IEEE arithmetic . . . . .	443
14.1	Derived types and constants defined in the modules . . . . .	444
14.2	The exceptions . . . . .	445
14.3	The rounding modes . . . . .	446
14.4	Underflow mode . . . . .	447
14.5	Halting . . . . .	447
14.6	The floating point status . . . . .	448
14.7	Exceptional values . . . . .	448
14.8	IEEE arithmetic . . . . .	448
14.9	Tables of the procedures . . . . .	449
14.9.1	Inquiry functions . . . . .	449
14.9.2	Elemental functions . . . . .	450
14.9.3	Kind function . . . . .	450
14.9.4	Elemental subroutines . . . . .	450
14.9.5	Nonelemental subroutines . . . . .	450
14.10	Specifications of the procedures . . . . .	451
14.11	Examples . . . . .	466
15	Interoperability with C . . . . .	471
15.1	General . . . . .	471
15.2	The ISO_C_BINDING intrinsic module . . . . .	471
15.2.1	Summary of contents . . . . .	471
15.2.2	Named constants and derived types in the module . . . . .	471
15.2.3	Procedures in the module . . . . .	473
15.3	Interoperability between Fortran and C entities . . . . .	476
15.3.1	General . . . . .	476
15.3.2	Interoperability of intrinsic types . . . . .	477
15.3.3	Interoperability with C pointer types . . . . .	479
15.3.4	Interoperability of derived types and C struct types . . . . .	480
15.3.5	Interoperability of scalar variables . . . . .	481
15.3.6	Interoperability of array variables . . . . .	481
15.3.7	Interoperability of procedures and procedure interfaces . . . . .	482
15.4	Interoperation with C global variables . . . . .	484
15.4.1	General . . . . .	484
15.4.2	Binding labels for common blocks and variables . . . . .	485
15.5	Interoperation with C functions . . . . .	485
15.5.1	Definition and reference of interoperable procedures . . . . .	485
15.5.2	Binding labels for procedures . . . . .	485
15.5.3	Exceptions and IEEE arithmetic procedures . . . . .	486
16	Scope, association, and definition . . . . .	487
16.1	Identifiers and entities . . . . .	487
16.2	Scope of global identifiers . . . . .	487
16.3	Scope of local identifiers . . . . .	488

16.3.1	Classes of local identifiers . . . . .	488
16.3.2	Local identifiers that are the same as common block names . . . . .	489
16.3.3	Function results . . . . .	489
16.3.4	Restrictions on generic declarations . . . . .	489
16.3.5	Components, type parameters, and bindings . . . . .	491
16.3.6	Argument keywords . . . . .	491
16.4	Statement and construct entities . . . . .	491
16.5	Association . . . . .	492
16.5.1	Name association . . . . .	492
16.5.2	Pointer association . . . . .	496
16.5.3	Storage association . . . . .	499
16.5.4	Inheritance association . . . . .	502
16.5.5	Establishing associations . . . . .	502
16.6	Definition and undefinition of variables . . . . .	503
16.6.1	Definition of objects and subobjects . . . . .	503
16.6.2	Variables that are always defined . . . . .	503
16.6.3	Variables that are initially defined . . . . .	503
16.6.4	Variables that are initially undefined . . . . .	504
16.6.5	Events that cause variables to become defined . . . . .	504
16.6.6	Events that cause variables to become undefined . . . . .	506
16.6.7	Variable definition context . . . . .	508
16.6.8	Pointer association context . . . . .	508
Annex A	(informative)Glossary of technical terms . . . . .	509
Annex B	(informative)Decremental features . . . . .	523
B.1	Deleted features . . . . .	523
B.2	Obsolescent features . . . . .	524
B.2.1	Alternate return . . . . .	524
B.2.2	Computed GO TO statement . . . . .	524
B.2.3	Statement functions . . . . .	524
B.2.4	DATA statements among executables . . . . .	525
B.2.5	Assumed character length functions . . . . .	525
B.2.6	Fixed form source . . . . .	525
B.2.7	CHARACTER* form of CHARACTER declaration . . . . .	525
Annex C	(informative)Extended notes . . . . .	527
C.1	Clause 4 notes . . . . .	527
C.1.1	Selection of the approximation methods (4.4.3) . . . . .	527
C.1.2	Type extension and component accessibility (4.5.2.2, 4.5.4) . . . . .	527
C.1.3	Abstract types . . . . .	528
C.1.4	Pointers (4.5.2) . . . . .	529
C.1.5	Structure constructors and generic names . . . . .	530
C.1.6	Generic type-bound procedures . . . . .	532
C.1.7	Final subroutines (4.5.6, 4.5.6.2, 4.5.6.3, 4.5.6.4) . . . . .	533
C.2	Clause 5 notes . . . . .	535
C.2.1	The POINTER attribute (5.3.13) . . . . .	535
C.2.2	The TARGET attribute (5.3.16) . . . . .	536
C.2.3	The VOLATILE attribute (5.3.18) . . . . .	536
C.3	Clause 6 notes . . . . .	537
C.3.1	Structure components (6.1.2) . . . . .	537
C.3.2	Allocation with dynamic type (6.3.1) . . . . .	538
C.3.3	Pointer allocation and association . . . . .	539
C.4	Clause 7 notes . . . . .	540

C.4.1	Character assignment . . . . .	540
C.4.2	Evaluation of function references . . . . .	540
C.4.3	Pointers in expressions . . . . .	540
C.4.4	Pointers on the left side of an assignment . . . . .	540
C.4.5	An example of a FORALL construct containing a WHERE construct . . . . .	541
C.4.6	Examples of FORALL statements . . . . .	542
C.5	Clause 8 notes . . . . .	543
C.5.1	Loop control . . . . .	543
C.5.2	The CASE construct . . . . .	543
C.5.3	Examples of DO constructs . . . . .	543
C.5.4	Examples of invalid DO constructs . . . . .	545
C.6	Clause 9 notes . . . . .	546
C.6.1	External files (9.2) . . . . .	546
C.6.2	Nonadvancing input/output (9.2.3.1) . . . . .	548
C.6.3	Asynchronous input/output . . . . .	549
C.6.4	OPEN statement (9.4.5) . . . . .	550
C.6.5	Connection properties (9.4.3) . . . . .	551
C.6.6	CLOSE statement (9.4.7) . . . . .	552
C.7	Clause 10 notes . . . . .	552
C.7.1	Number of records (10.4, 10.5, 10.8.2) . . . . .	552
C.7.2	List-directed input (10.10.1) . . . . .	553
C.8	Clause 11 notes . . . . .	553
C.8.1	Main program and block data program unit (11.1, 11.3) . . . . .	553
C.8.2	Dependent compilation (11.2) . . . . .	554
C.8.3	Examples of the use of modules . . . . .	556
C.8.4	Modules with submodules . . . . .	562
C.9	Clause 12 notes . . . . .	567
C.9.1	Portability problems with external procedures (12.4.2.2) . . . . .	567
C.9.2	Procedures defined by means other than Fortran (12.6.3) . . . . .	567
C.9.3	Procedure interfaces (12.4) . . . . .	567
C.9.4	Abstract interfaces (12.4) and procedure pointer components (4.5) . . . . .	568
C.9.5	Argument association and evaluation (12.5.1.4) . . . . .	570
C.9.6	Pointers and targets as arguments (12.5.1.4) . . . . .	571
C.9.7	Polymorphic Argument Association (12.5.1.6) . . . . .	572
C.10	Clause 13 notes . . . . .	573
C.10.1	Module for THIS_IMAGE and IMAGE_INDEX . . . . .	573
C.10.2	Collective co-array subroutine variations . . . . .	574
C.11	Clause 15 notes . . . . .	575
C.11.1	Runtime environments . . . . .	575
C.11.2	Examples of Interoperation between Fortran and C Functions . . . . .	575
C.12	Clause 16 notes . . . . .	580
C.12.1	Examples of host association (16.5.1.4) . . . . .	580
C.12.2	Rules ensuring unambiguous generics (16.3.4) . . . . .	581
C.13	Array feature notes . . . . .	586
C.13.1	Summary of features . . . . .	586
C.13.2	Examples . . . . .	588
C.13.3	FORMula TRANslation and array processing . . . . .	592
C.13.4	Sum of squared residuals . . . . .	593
C.13.5	Vector norms: infinity-norm and one-norm . . . . .	593
C.13.6	Matrix norms: infinity-norm and one-norm . . . . .	593
C.13.7	Logical queries . . . . .	594
C.13.8	Parallel computations . . . . .	594
C.13.9	Example of element-by-element computation . . . . .	595
C.13.10	Bit manipulation and inquiry procedures . . . . .	595

Annex D	(informative)Syntax rules . . . . .	597
D.1	Extract of all syntax rules . . . . .	597
D.2	Syntax rule cross-reference . . . . .	641
Annex E	(informative)Index . . . . .	657



## List of Tables

2.1	Requirements on statement ordering . . . . .	14
2.2	Statements allowed in scoping units . . . . .	15
3.1	Special characters . . . . .	26
6.1	Subscript order value . . . . .	121
7.1	Type of operands and results for intrinsic operators . . . . .	137
7.2	Interpretation of the numeric intrinsic operators . . . . .	151
7.3	Interpretation of the character intrinsic operator // . . . . .	151
7.4	Interpretation of the relational intrinsic operators . . . . .	152
7.5	Interpretation of the logical intrinsic operators . . . . .	153
7.6	The values of operations involving logical intrinsic operators . . . . .	153
7.7	Interpretation of the bits intrinsic operators . . . . .	154
7.8	The values of bitwise operations involving bits intrinsic operators . . . . .	154
7.9	Categories of operations and relative precedence . . . . .	154
7.10	Type conformance for the intrinsic assignment statement . . . . .	158
7.11	Numeric conversion and the assignment statement . . . . .	160
7.12	Bits conversion and the assignment statement . . . . .	160
13.1	Characteristics of the result of NULL ( ) . . . . .	413
15.1	Names of C characters with special semantics . . . . .	472
15.2	Interoperability between Fortran and C types . . . . .	477

## 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 the standard.

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 BOZ constants.

### 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

# 1 Information technology — Programming languages — 2 Fortran —

## 3 Part 1: 4 Base Language

### 5 1 Overview

#### 6 1.1 Scope

7 ISO/IEC 1539 is a multipart International Standard; the parts are published separately. This publi-  
8 cation, ISO/IEC 1539-1, which is the first part, specifies the form and establishes the interpretation  
9 of programs expressed in the base Fortran language. The purpose of this part of ISO/IEC 1539 is to  
10 promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on  
11 a variety of computing systems. The second part, ISO/IEC 1539-2, defines additional facilities for the  
12 manipulation of character strings of variable length. The third part, ISO/IEC 1539-3, defines a stan-  
13 dard conditional compilation facility for Fortran. A processor conforming to part 1 need not conform to  
14 ISO/IEC 1539-2 or ISO/IEC 1539-3; however, conformance to either assumes conformance to this part.  
15 Throughout this publication, the term “this standard” refers to ISO/IEC 1539-1.

#### 16 1.2 Processor

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

#### 19 1.3 Inclusions

20 This part of ISO/IEC 1539 specifies

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

#### 25 1.4 Exclusions

26 This part of ISO/IEC 1539 does not specify

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

- 1 (8) of 1.5,
- 2 (5) the size or complexity of a program and its data that will exceed the capacity of any
- 3 particular computing system or the capability of a particular processor,
- 4 (6) the physical properties of the representation of quantities and the method of rounding,
- 5 approximating, or computing numeric values on a particular processor,
- 6 (7) the physical properties of input/output records, files, and units, or
- 7 (8) the physical properties and implementation of storage.

## 8 1.5 Conformance

9 A program (2.2.1) is a **standard-conforming program** if it uses only those forms and relationships  
10 described herein and if the program has an interpretation according to this part of ISO/IEC 1539. A  
11 program unit (2.2) conforms to this part of ISO/IEC 1539 if it can be included in a program in a manner  
12 that allows the program to be standard conforming.

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

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

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

35 A standard-conforming processor may allow additional forms and relationships provided that such ad-  
36 ditions do not conflict with the standard forms and relationships. However, a standard-conforming  
37 processor may allow additional intrinsic procedures even though this could cause a conflict with the  
38 name of a procedure in a standard-conforming program. If such a conflict occurs and involves the name  
39 of an external procedure, the processor is permitted to use the intrinsic procedure unless the name is  
40 given the EXTERNAL attribute (5.3.8) in the scoping unit (16). A standard-conforming program shall  
41 not use nonstandard intrinsic procedures or modules that have been added by the processor.

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

- 1 In some cases, this part of ISO/IEC 1539 allows the provision of facilities that are not completely specified  
2 in the standard. These facilities are identified as **processor dependent**. They shall be provided, with  
3 methods or 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.

## 4 **1.6 Compatibility**

### 5 **1.6.1 New intrinsic procedures**

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

### 11 **1.6.2 New intrinsic data type and operator**

- 12 This part of ISO/IEC 1539 specifies a new intrinsic type, BITS, which will conflict with a derived type  
13 of the same name.
- 14 This part of ISO/IEC 1539 specifies a new intrinsic operator, .XOR., which might conflict with a user-  
15 defined operator of the same name, and has a different precedence than in previous revisions of this part  
16 of ISO/IEC 1539.

### 17 **1.6.3 Fortran 2003 compatibility**

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

### 22 **1.6.4 Fortran 95 compatibility**

- 23 Except as identified in this subclause, this part of ISO/IEC 1539 is an upward compatible extension to  
24 ISO/IEC 1539-1:1997 (Fortran 95). Any standard-conforming Fortran 95 program remains standard-  
25 conforming under this part of ISO/IEC 1539. The following Fortran 95 features may have different  
26 interpretations in this part of ISO/IEC 1539.

- 27 (1) Earlier Fortran standards had the concept of printing, meaning that column one of formatted  
28 output had special meaning for a processor-dependent (possibly empty) set of external  
29 files. This could be neither detected nor specified by a standard-specified means. The  
30 interpretation of the first column is not specified by this part of ISO/IEC 1539.
- 31 (2) This part of ISO/IEC 1539 specifies a different output format for real zero values in list-  
32 directed and namelist output.

- 1       (3) If the processor can distinguish between positive and negative real zero, this part of ISO/IEC  
2       1539 requires different returned values for ATAN2(Y,X) when  $X < 0$  and Y is negative real  
3       zero and for LOG(X) and SQRT(X) when X is complex with  $\text{REAL}(X) < 0$  and negative  
4       zero imaginary part.

### 5   **1.6.5 Fortran 90 compatibility**

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

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

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

### 15   **1.6.6 FORTRAN 77 compatibility**

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

- 25       (1) FORTRAN 77 permitted a processor to supply more precision derived from a real constant  
26       than can be represented in a real datum when the constant is used to initialize a data object  
27       of type double precision real in a DATA statement. This part of ISO/IEC 1539 does not  
28       permit a processor this option.
- 29       (2) If a named variable that was not in a common block was initialized in a DATA statement and  
30       did not have the SAVE attribute specified, FORTRAN 77 left its SAVE attribute processor  
31       dependent. This part of ISO/IEC 1539 specifies (5.4.6) that this named variable has the  
32       SAVE attribute.
- 33       (3) FORTRAN 77 specified that the number of characters required by the input list was to be  
34       less than or equal to the number of characters in the record during formatted input. This  
35       part of ISO/IEC 1539 specifies (9.5.3.4.2) that the input record is logically padded with  
36       blanks if there are not enough characters in the record, unless the PAD= specifier with the  
37       value 'NO' is specified in an appropriate OPEN or READ statement.
- 38       (4) A value of 0 for a list item in a formatted output statement will be formatted in a different  
39       form for some G edit descriptors. In addition, this part of ISO/IEC 1539 specifies how  
40       rounding of values will affect the output field form, but FORTRAN 77 did not address this  
41       issue. Therefore, some FORTRAN 77 processors may produce an output form different from  
42       the output form produced by Fortran 2003 processors for certain combinations of values and  
43       G edit descriptors.
- 44       (5) If the processor can distinguish between positive and negative real zero, the behavior of the  
45       SIGN intrinsic function when the second argument is negative real zero is changed by this  
46       part of ISO/IEC 1539.

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

### 1.7.1 Applicability of requirements

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

### 1.7.2 Informative notes

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

### 1.7.3 Syntax rules

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

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

Common abbreviations used in syntactic terms are:

<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

- (3) The syntactic metasympols used are:

<b>is</b>	introduces a syntactic class definition
<b>or</b>	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

- (4) Each syntax rule is given a unique identifying number of the form Rsnn, where s is a one- or two-digit clause number and nn is a two-digit sequence number within that clause. The syntax rules are distributed as appropriate throughout the text, and are referenced by number as needed. Some rules in Clauses 2 and 3 are more fully described in later clauses; in such cases, the clause number s is the number of the later clause where the rule is repeated.
- (5) The syntax rules are not a complete and accurate syntax description of Fortran, and cannot be used to generate a Fortran parser automatically; where a syntax rule is incomplete, it is restricted by corresponding constraints and text.

#### NOTE 1.2

An example of the use of the syntax rules is:

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

**NOTE 1.2 (cont.)**

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

*digit*  
*digit digit*  
*digit digit digit digit*  
*digit digit digit digit digit digit digit digit*

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

4  
67  
1999  
10243852

**1.7.4 Constraints**

Each constraint is given a unique identifying number of the form C<sub>snn</sub>, where s is a one- or two-digit clause number and nn is a two-digit sequence number within that clause.

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

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

**1.7.5 Assumed syntax rules**

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

R101    *xyz-list*                                **is**   *xyz* [ , *xyz* ] ...  
R102    *xyz-name*                                **is**   *name*  
R103    *scalar-xyz*                               **is**   *xyz*

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

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

**1.7.6 Syntax conventions and characteristics**

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

(2) The rules on statement ordering are described rigorously in the definition of *program-unit* (R202). Expression hierarchy is described rigorously in the definition of *expr* (R722).



- 1 (3) The suffix “-*spec*” is used consistently for specifiers, such as input/output statement speci-  
2 fiers. It also is used for type declaration attribute specifications (for example, “*array-spec*”  
3 in R510), and in a few other cases.
- 4 (4) Where reference is made to a type parameter, including the surrounding parentheses, the  
5 suffix “-*selector*” is used. See, for example, “*kind-selector*” (R405) and “*length-selector*”  
6 (R421).
- 7 (5) The term “*subscript*” (for example, R619, R620, and R621) is used consistently in array  
8 definitions.

### 9 1.7.7 Text conventions

10 In the descriptive text, an equivalent English word is frequently used in place of a syntactic term.  
11 Particular statements and attributes are identified in the text by an upper-case keyword, e.g., “END  
12 statement”. Boldface words are used in the text where they are first defined with a specialized meaning.  
13 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.

## 14 1.8 Deleted and obsolescent features

### 15 1.8.1 General

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

### 22 1.8.2 Nature of deleted features

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

### 25 1.8.3 Nature of obsolescent features

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

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

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

## 32 1.9 Normative references

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

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

- 1 ISO 8601:1988, *Data elements and interchange formats—Information interchange—*  
2 *Representation of dates and times.*
- 3 ISO/IEC 9899:1999, *Information technology—Programming languages—C.*
- 4 ISO/IEC 10646-1:2000, *Information technology—Universal multiple-octet coded character set (UCS)—*  
5 *Part 1: Architecture and basic multilingual plane.*
- 6 IEC 60559 (1989-01), *Binary floating-point arithmetic for microprocessor systems.*
- 7 ISO/IEC 646:1991 (International Reference Version) is the international equivalent of ANSI X3.4-1986,  
8 commonly known as ASCII.
- 9 This part of ISO/IEC 1539 refers to ISO/IEC 9899:1999 as the C International Standard.
- 10 Because IEC 60559 (1989-01) was originally IEEE 754-1985, *Standard for binary floating-point arith-*  
11 *metic*, and is widely known by this name, this standard refers to it as the IEEE International 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 standard 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*

R1225 *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*  
[ *specification-part* ]  
[ *module-subprogram-part* ]  
*end-submodule-stmt*

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

1			<i>end-block-data-stmt</i>
2	R204	<i>specification-part</i>	<b>is</b> [ <i>use-stmt</i> ] ...
3			[ <i>import-stmt</i> ] ...
4			[ <i>implicit-part</i> ]
5			[ <i>declaration-construct</i> ] ...
6	R205	<i>implicit-part</i>	<b>is</b> [ <i>implicit-part-stmt</i> ] ...
7			<i>implicit-stmt</i>
8	R206	<i>implicit-part-stmt</i>	<b>is</b> <i>implicit-stmt</i>
9			<b>or</b> <i>parameter-stmt</i>
10			<b>or</b> <i>format-stmt</i>
11			<b>or</b> <i>entry-stmt</i>
12	R207	<i>declaration-construct</i>	<b>is</b> <i>derived-type-def</i>
13			<b>or</b> <i>entry-stmt</i>
14			<b>or</b> <i>enum-def</i>
15			<b>or</b> <i>format-stmt</i>
16			<b>or</b> <i>interface-block</i>
17			<b>or</b> <i>macro-definition</i>
18			<b>or</b> <i>parameter-stmt</i>
19			<b>or</b> <i>procedure-declaration-stmt</i>
20			<b>or</b> <i>specification-stmt</i>
21			<b>or</b> <i>type-declaration-stmt</i>
22			<b>or</b> <i>stmt-function-stmt</i>
23	R208	<i>execution-part</i>	<b>is</b> <i>executable-construct</i>
24			[ <i>execution-part-construct</i> ] ...
25	R209	<i>execution-part-construct</i>	<b>is</b> <i>executable-construct</i>
26			<b>or</b> <i>format-stmt</i>
27			<b>or</b> <i>entry-stmt</i>
28			<b>or</b> <i>data-stmt</i>
29	R210	<i>internal-subprogram-part</i>	<b>is</b> <i>contains-stmt</i>
30			[ <i>internal-subprogram</i> ] ...
31	R211	<i>internal-subprogram</i>	<b>is</b> <i>function-subprogram</i>
32			<b>or</b> <i>subroutine-subprogram</i>
33	R1107	<i>module-subprogram-part</i>	<b>is</b> <i>contains-stmt</i>
34			[ <i>module-subprogram</i> ] ...
35	R1108	<i>module-subprogram</i>	<b>is</b> <i>function-subprogram</i>
36			<b>or</b> <i>subroutine-subprogram</i>
37			<b>or</b> <i>separate-module-subprogram</i>
38	R212	<i>specification-stmt</i>	<b>is</b> <i>access-stmt</i>
39			<b>or</b> <i>allocatable-stmt</i>
40			<b>or</b> <i>asynchronous-stmt</i>
41			<b>or</b> <i>bind-stmt</i>
42			<b>or</b> <i>common-stmt</i>
43			<b>or</b> <i>data-stmt</i>
44			<b>or</b> <i>dimension-stmt</i>
45			<b>or</b> <i>equivalence-stmt</i>
46			<b>or</b> <i>external-stmt</i>
47			<b>or</b> <i>intent-stmt</i>
48			<b>or</b> <i>intrinsic-stmt</i>
49			<b>or</b> <i>namelist-stmt</i>
50			<b>or</b> <i>optional-stmt</i>
51			<b>or</b> <i>pointer-stmt</i>
52			<b>or</b> <i>protected-stmt</i>
53			<b>or</b> <i>save-stmt</i>
54			<b>or</b> <i>target-stmt</i>

1		<b>or</b> <i>volatile-stmt</i>
2		<b>or</b> <i>value-stmt</i>
3	R213 <i>executable-construct</i>	<b>is</b> <i>action-stmt</i>
4		<b>or</b> <i>associate-construct</i>
5		<b>or</b> <i>block-construct</i>
6		<b>or</b> <i>case-construct</i>
7		<b>or</b> <i>critical-construct</i>
8		<b>or</b> <i>do-construct</i>
9		<b>or</b> <i>forall-construct</i>
10		<b>or</b> <i>if-construct</i>
11		<b>or</b> <i>select-type-construct</i>
12		<b>or</b> <i>where-construct</i>
13	R214 <i>action-stmt</i>	<b>is</b> <i>allocate-stmt</i>
14		<b>or</b> <i>assignment-stmt</i>
15		<b>or</b> <i>backspace-stmt</i>
16		<b>or</b> <i>call-stmt</i>
17		<b>or</b> <i>close-stmt</i>
18		<b>or</b> <i>continue-stmt</i>
19		<b>or</b> <i>cycle-stmt</i>
20		<b>or</b> <i>deallocate-stmt</i>
21		<b>or</b> <i>endfile-stmt</i>
22		<b>or</b> <i>end-function-stmt</i>
23		<b>or</b> <i>end-program-stmt</i>
24		<b>or</b> <i>end-subroutine-stmt</i>
25		<b>or</b> <i>exit-stmt</i>
26		<b>or</b> <i>flush-stmt</i>
27		<b>or</b> <i>forall-stmt</i>
28		<b>or</b> <i>goto-stmt</i>
29		<b>or</b> <i>if-stmt</i>
30		<b>or</b> <i>inquire-stmt</i>
31		<b>or</b> <i>notify-stmt</i>
32		<b>or</b> <i>nullify-stmt</i>
33		<b>or</b> <i>open-stmt</i>
34		<b>or</b> <i>pointer-assignment-stmt</i>
35		<b>or</b> <i>print-stmt</i>
36		<b>or</b> <i>query-stmt</i>
37		<b>or</b> <i>read-stmt</i>
38		<b>or</b> <i>return-stmt</i>
39		<b>or</b> <i>rewind-stmt</i>
40		<b>or</b> <i>stop-stmt</i>
41		<b>or</b> <i>sync-all-stmt</i>
42		<b>or</b> <i>sync-images-stmt</i>
43		<b>or</b> <i>sync-memory-stmt</i>
44		<b>or</b> <i>sync-team-stmt</i>
45		<b>or</b> <i>wait-stmt</i>
46		<b>or</b> <i>where-stmt</i>
47		<b>or</b> <i>write-stmt</i>
48		<b>or</b> <i>arithmetic-if-stmt</i>
49		<b>or</b> <i>computed-goto-stmt</i>
50	C201    (R208) An <i>execution-part</i> shall not contain an <i>end-function-stmt</i> , <i>end-program-stmt</i> , or <i>end-</i>	
51	<i>subroutine-stmt</i> .	
52	Additionally, an EXPAND statement may occur anywhere that any statement may occur other than	

- 1 as the first statement of a program unit. The syntax rules are applied to the program after macro  
2 expansion, i.e. with each EXPAND statement replaced by the statements it produces.

## 3 2.2 Program unit concepts

4 Program units are the fundamental components of a Fortran program. A **program unit** may be a main  
5 program, an external subprogram, a module, a submodule, or a block data program unit. A subprogram  
6 may be a function subprogram or a subroutine subprogram. A module contains definitions that are to  
7 be made accessible to other program units. A submodule is an extension of a module; it may contain  
8 the definitions of procedures declared in a module or another submodule. A block data program unit is  
9 used to specify initial values for data objects in named common blocks. Each type of program unit is  
10 described in Clause 11 or 12. An **external subprogram** is a subprogram that is not in a main program,  
11 a module, a submodule, or another subprogram. An **internal subprogram** is a subprogram that is in  
12 a main program or another subprogram. A **module subprogram** is a subprogram that is in a module  
13 or submodule but is not an internal subprogram.

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

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

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

### 20 2.2.1 Program

21 A **program** consists of exactly one main program, any number (including zero) of other kinds of program  
22 units, and any number (including zero) of external procedures and other entities defined by means other  
23 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 standard 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 (11.2.1) is processed, a processor may require a particular order of processing of the program units.

### 24 2.2.2 Main program

25 The Fortran main program is described in 11.1.

### 26 2.2.3 Procedure

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

1 Procedures are described further in Clause 12.

#### 2 2.2.3.1 External procedure

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

#### 6 2.2.3.2 Module procedure

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

#### 9 2.2.3.3 Internal procedure

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

#### 14 2.2.3.4 Interface block

15 An **interface body** describes an abstract interface or the interface of a dummy procedure, external  
16 procedure, procedure pointer, or type-bound procedure.

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

- 20 (1) by using a generic name,
- 21 (2) by using a defined operator,
- 22 (3) by using a defined assignment, or
- 23 (4) for derived-type input/output.

#### 24 2.2.4 Module

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

#### 29 2.2.5 Submodule

30 A **submodule** is a program unit that extends a module or another submodule. It may provide definitions  
31 (12.6) for procedures whose interfaces are declared (12.4.2.1) in an ancestor module or submodule. It  
32 may also contain declarations and definitions of other entities, which are accessible in its descendants.  
33 An entity declared in a submodule is not accessible by use association unless it is a module procedure  
34 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.
------------------------------------------------------------------------------------------------------------------

## 2.3 Execution concepts

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

### 2.3.1 Executable/nonexecutable statements

- Program 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*.

- 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.

### 2.3.2 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**

Kind of scoping unit:	Main program	Module or submodule	Block data	External subprog	Module subprog	Internal subprog	Interface body
USE statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IMPORT statement	No	No	No	No	No	No	Yes
ENTRY statement	No	No	No	Yes	Yes	No	No
FORMAT statement	Yes	No	No	Yes	Yes	Yes	No
Misc. decls (see note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DATA statement	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type definition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface block	Yes	Yes	No	Yes	Yes	Yes	Yes
Executable statement	Yes	No	No	Yes	Yes	Yes	No
CONTAINS statement	Yes	Yes	No	Yes	Yes	No	No
Statement function stmt.	Yes	No	No	Yes	Yes	Yes	No
Notes for Table 2.2: 1) Misc. declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, enumeration definitions, procedure declaration statements, and specification statements. 2) The scoping unit of a module or submodule does not include any module subprograms that it contains.							

### 2.3.3 The END statement

An *end-program-stmt*, *end-function-stmt*, *end-subroutine-stmt*, *end-sep-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-sep-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-sep-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.

### 2.3.4 Execution sequence

If a program contains a Fortran main program, execution of the program begins 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 procedure is invoked, execution begins with the first executable construct appearing 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) END=, ERR=, and EOR= specifiers may result in a branch.
- (4) Alternate returns may result in a branch.

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

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

3 Normal termination of execution of the program occurs on all images if a STOP statement or *end-*  
4 *program-stmt* is executed immediately after a SYNC\_ALL (8.5.2) statement on all images. The stop  
5 code, if any, and warnings of any exceptions that are signaling (8.4) shall be made available only for  
6 image 1.

### J3 internal note

#### Unresolved Technical Issue 3

Since there is only one program, I do not see how "Normal termination of execution" can be qualified by "occurs on all images". If execution of the program is terminated, since that execution is all of the image executions, each image is terminated. This is straightforward from the first sentence of 2.3.5.

It was suggested to me that the issue was whether termination was "normal" on all images; well, if it is normal for the whole program, it is normal on all images.

### J3 internal note

#### Unresolved Technical Issue 4

"shall be made available only for image 1" is problematic since after normal termination no images exist. I do not understand what is meant here. Maybe (?) it is that any signalling exception on any image other than 1 is not reported?"

Subgroup says "This is in direct response to complaints about MPI programs printing "Fortran STOP" to the screen 10000 times."

The editor says: There is nothing in the standard which prevents a processor from printing "Fortran STOP" to the screen 10000 times even in single-process Fortran 77. I think that is purely and simply a matter of QoI (or possibly even outwith the scope). If we want to say something here it should almost certainly be \*a note\* along the lines that since a program is a program, however many "images" it contains, that processors should be circumspect about reporting identical statuses multiple times on image termination.

Anyway, failure to report signalling exceptions just because they are on images other than the image numbered "1" is poor design. The consequences of overflow and invalid operations are just as potentially serious whether the computation takes part on image 1 or some other image. It would be preferable not to disallow a conscientious processor from reporting such problems.

Finally, this is a small detail. There is currently no mention of what happens with signalling exceptions in clause 2, and so no need to mention how it differs with co-arrays. Therefore the second sentence should be deleted.

7 When a STOP or *end-program-stmt* that does not immediately follow a SYNC\_ALL statement is executed  
8 on an image, normal termination occurs on that image. The stop code, if any, and warnings of any  
9 exceptions that are signaling shall be made available only for that image. The executions of all other  
10 images are terminated.

**J3 internal note**

## Unresolved Technical Issue 5

This is inconsistent with the behaviour after SYNC\_ALL (unless image 1 always restarts first after a SYNC\_ALL). This is going to cause headaches for the implementors (STOP has multiple meanings, and does different things according to context - ugh).

There seems to be two competing theories for what STOP is for here: (1) a programmer-initiated error termination (2) a normal termination and the decision as to which STOP is which kind is being determined by whether it is preceded by SYNC\_ALL (not that this is adequately defined, see J3 notes in the STOP statement section).

In previous Fortrans, STOP is normal termination. That's it. It is not "CALL ABORT". If we want a second kind of STOP, perhaps we should consider adding a second kind of STOP, perhaps spelled QUICKSTOP or something.

And of course the wording is bad ("made available") but wordsmithing can wait until the text is fixed or deleted.

**NOTE 2.4**

When a STOP or *end-program-stmt* statement is executed on an image, how soon termination takes place on another image is processor dependent.

**J3 internal note**

## Unresolved Technical Issue 6

Presumably this is referring to the "error STOP" (or the "STOP not after a SYNC\_ALL") situation, since it does not appear to be processor-dependent in the "normal STOP" case. Or perhaps it is... again, this doesn't seem to be well-defined."

1 **2.3.5 Images**

- 2 A Fortran program executes as if it were replicated a number of times (which may be one), the number  
3 of replications remaining fixed during execution of the program.

**J3 internal note**

## Unresolved Technical Issue 1

In post-draft discussions it has been variously and contradictorily contended that the concept of program does not include all the images. It is certainly contradictory in the edits.

Look, if we \*have\* a definition of single-threaded Fortran (like we used to) then we can layer \*\*\*SOMETHING ELSE\*\*\* on top of it and replicate as we please without affecting the single-threaded concepts. But tightly integrating them does not give us this ability. If "program" is meant to include only the execution and data entities of a single image, we still need a name for the concept that is the "Co-Fortran program" providing the semantics for the whole "Co-Fortran" thing.

- 4 Each copy is called an **image** and has its own set of data objects and procedure pointers. Each image  
5 executes asynchronously.

**NOTE 2.5**

The programmer controls the progress of execution in each image through explicit use of Fortran control constructs (8.1, 8.2). The 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.

**J3 internal note****Unresolved Technical Issue 2**

Missing from the "own set" is execution state, however we say it.

It has been contended that all of that is implicit in "executes asynchronously", but since there are many points of non-asynchronous behaviour - e.g. in image termination effects - just how far that extends is emphatically NOT obvious. IEEE modes, external units, file system, we all need to know. Otherwise it is not standardised (a situation we should all hope to avoid). Either we should say that "All other entities are shared between the images" (after putting execution state into the "own" set!), or we should specify the entities that are shared; at least then any entity that does not fall into either list can be seen to be a problem... I suppose I recommend the latter course. Once this is fixed the edit at [178:36] can be turned into a note instead. (Actually, I think the whole paragraph at 178:35-36 should probably be inside the note, since we already say this under "scoping".)

- 1 Each image is identified by an **image index**, which is an integer value in the range one to the number
- 2 of images.
- 3 A processor shall contain the capability of executing a program on a single image.

**NOTE 2.6**

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.

- 4 A **team** is a set of images formed by invoking the intrinsic collective subroutine FORM\_TEAM (13.7.69)
- 5 for the purposes of collaboration. A team is identified by a scalar variable of type IMAGE\_TEAM
- 6 (13.8.3.5).

## 7 **2.4 Data concepts**

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

### 10 **2.4.1 Type**

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

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

- 17 There are two categories of types: intrinsic types and derived types.

#### 18 **2.4.1.1 Intrinsic type**

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

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

### 1 2.4.1.2 Derived type

2 A **derived type** is a type that is defined by a type definition or by an intrinsic module. A scalar object of  
 3 derived type is called a **structure** (4.5). Derived types may be parameterized. Assignment of structures  
 4 is defined intrinsically (7.4.1.3), but there are no intrinsic operations for structures. For each derived  
 5 type, a structure constructor is available to provide values (4.5.10). In addition, data objects of derived  
 6 type may be used as procedure arguments and function results, and may appear in input/output lists.  
 7 If additional operations are needed for a derived type, they shall be supplied as procedure definitions.

8 Derived types are described further in 4.5.

### 9 2.4.2 Data value

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

### 14 2.4.3 Data entity

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

#### 19 2.4.3.1 Data object

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

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

29 Objects referenced by a name are:

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

31 Subobjects referenced by an object designator are:

32	an array element	(a scalar subobject)
	an array section	(an array subobject)
	a structure component	(a scalar or an array subobject)
	a substring	(a scalar subobject)

#### 33 2.4.3.1.1 Variable

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

35 A named **local variable** of the scoping unit of a module, submodule, main program, or subprogram, is a  
 36 named variable that is a local entity of the scoping unit, is not a dummy argument, is not in COMMON,

1 does not have the BIND attribute, and is not accessed by use or host association. A named local variable  
 2 of a BLOCK construct is a named variable that is declared in that construct, is not in COMMON, does  
 3 not have the BIND attribute, and is not accessed by use association. A subobject of a named local  
 4 variable is also a local variable.

#### 5 2.4.3.1.2 Constant

6 A **constant** has a value and cannot become defined, redefined, or undefined during execution of a  
 7 program. A constant with a name is called a **named constant** and has the PARAMETER attribute  
 8 (5.3.12). A constant without a name is called a **literal constant** (4.4).

#### 9 2.4.3.1.3 Subobject of a constant

10 A **subobject of a constant** is a portion of a constant. The portion referenced may depend on the  
 11 value of a variable.

#### NOTE 2.7

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.

#### 12 2.4.3.2 Expression

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

#### 16 2.4.3.3 Function reference

17 A **function reference** (12.5.2) produces a data entity when the function is executed during expression  
 18 evaluation. The type, type parameters, and rank of a function result are determined by the interface of  
 19 the function (12.3.2). The value of a function result is determined by execution of the function.

#### 20 2.4.4 Scalar

21 A **scalar** is a datum that is not an array. Scalars may be of any type.

#### NOTE 2.8

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

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

#### 23 2.4.5 Array

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

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

8 Two arrays are **conformable** if they have the same shape. A scalar is conformable with any array. Any  
 9 intrinsic operation defined for scalar objects may be applied to conformable objects. Such operations  
 10 are performed element-by-element to produce a resultant array conformable with the array operands.  
 11 Element-by-element operation means corresponding elements of the operand arrays are involved in a  
 12 scalar operation to produce the corresponding element in the result array. Such an operation is described  
 13 as **elemental**.

#### NOTE 2.9

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.

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

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

### 17 2.4.6 Co-array

18 A **co-array** is a data entity that is declared with nonzero co-rank and may be referenced or defined by  
 19 any image. It may be a scalar or an array. For each co-array on an image, there is a corresponding  
 20 co-array with the same type, type parameters, and bounds on every other image. The co-array on any  
 21 image may be accessed by using *co-subscripts*. On its own image, a co-array may be accessed without  
 22 use of co-subscripts.

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

#### J3 internal note

##### Unresolved Technical Issue 7

The term "co-dimension" is not defined. It might be better to define co-array in terms of being a rectangular set of objects in co-rank "co-dimensions" etc., similarly to how we define arrays.

25 An object whose designator includes an *image-selector* is a **co-indexed object**. For each co-array,  
 26 co-subscripts are mapped to image indices in the same way as Fortran array subscripts are mapped to  
 27 the position of the array element in array element order (6.2.2.2).

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

#### NOTE 2.10

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.

## 1 2.4.7 Pointer

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

4 A pointer is **associated** with a **target** by pointer assignment (7.4.2). A data pointer may also be  
5 associated with a target by allocation (6.3.1). A pointer is **disassociated** following execution of a  
6 NULLIFY statement, following pointer assignment with a disassociated pointer, by default initialization,  
7 or by explicit initialization. A data pointer may also be disassociated by execution of a DEALLOCATE  
8 statement. A disassociated pointer is not associated with a target (16.5.2).

9 A pointer that is not associated shall not be referenced or defined.

10 If a data pointer is an array, the rank is declared, but the extents are determined when the pointer is  
11 associated with a target.

## 12 2.4.8 Storage

13 Many of the facilities of this standard make no assumptions about the physical storage characteristics of  
14 data objects. However, program units that include storage association dependent features shall observe  
15 the storage restrictions described in 16.5.3.

## 16 2.5 Fundamental terms

17 For the purposes of this standard, the definitions of terms in this subclause apply.

### 18 2.5.1 Name and designator

19 A **name** is used to identify a program constituent, such as a program unit, named variable, named  
20 constant, dummy argument, or derived type. The rules governing the construction of names are given  
21 in 3.2.1. A **designator** is a name followed by zero or more component selectors, complex part selectors,  
22 array section selectors, array element selectors, image selectors, and substring selectors.

23 An **object designator** is a designator for a data object. A **procedure designator** is a designator for  
24 a procedure.

#### NOTE 2.11

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

### 25 2.5.2 Keyword

26 The term **keyword** is used in two ways.

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



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

#### NOTE 2.12

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 2.5.3 Association

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

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

### 9 2.5.4 Declaration

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

### 12 2.5.5 Definition

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

- 14 (1) It refers to the specification of derived types and procedures.
- 15 (2) When an object is given a valid value during program execution, it is said to become  
 16 **defined**. This is often accomplished by execution of an assignment or input statement.  
 17 When a variable does not have a predictable value, it is said to be **undefined**. Similarly,  
 18 when a pointer is associated with a target or nullified, its pointer association status is said  
 19 to become **defined**. When the association status of a pointer is not predictable, its pointer  
 20 association status is said to be **undefined**.

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

### 22 2.5.6 Reference

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

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

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

31 A **module reference** is the appearance of a module name in a USE statement (11.2.1).

### 32 2.5.7 Intrinsic

33 The qualifier **intrinsic** has two meanings.

- 1       (1) The qualifier signifies that the term to which it is applied is defined in this standard. In-  
2       intrinsic applies to types, procedures, modules, assignment statements, and operators. All  
3       intrinsic types, procedures, assignments, and operators may be used in any scoping unit  
4       without further definition or specification. Intrinsic modules may be accessed by use as-  
5       sociation. Intrinsic procedures and modules defined in this standard are called standard  
6       intrinsic procedures and standard intrinsic modules, respectively.
- 7       (2) The qualifier applies to procedures or modules that are provided by a processor but are not  
8       defined in this standard (13, 14, 15.2). Such procedures and modules are called nonstandard  
9       intrinsic procedures and nonstandard intrinsic modules, respectively.

## 10 2.5.8 Operator

11 An **operator** specifies a computation involving one (unary operator) or two (binary operator) data values  
12 (**operands**). This standard specifies a number of intrinsic operators (e.g., the arithmetic operators +, −,  
13 \*, /, and \*\* with numeric operands and the logical operators .AND., .OR., etc. with logical operands).  
14 Additional operators may be defined within a program (7.1.3).

## 15 2.5.9 Sequence

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

18 The elements of a nonempty sequence are referred to as the first element, second element, etc. The  
19  $n$ th element, where  $n$  is the number of elements in the sequence, is called the last element. An empty  
20 sequence has no first or last element.

## 21 2.5.10 Companion processors

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

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

### NOTE 2.13

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.

## 1 3 Characters, lexical tokens, and source form

2 This clause describes the Fortran character set and the various lexical tokens such as names and operators.

3 This clause also describes the rules for the forms that Fortran programs may take.

### 4 3.1 Processor character set

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

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

9 R301	<i>character</i>	<b>is</b>	<i>alphanumeric-character</i>
10		<b>or</b>	<i>special-character</i>
11 R302	<i>alphanumeric-character</i>	<b>is</b>	<i>letter</i>
12		<b>or</b>	<i>digit</i>
13		<b>or</b>	<i>underscore</i>

14 Except for the currency symbol, the graphics used for the characters shall be as given in 3.1.1, 3.1.2,  
15 3.1.3, and 3.1.4. However, the style of any graphic is not specified.

16 The default character type shall support a character set that includes the Fortran character set. By sup-  
17 plying nondefault character types, the processor may support additional character sets. The characters  
18 available in the ASCII and ISO 10646 character sets are specified by ISO/IEC 646:1991 (International  
19 Reference Version) and ISO/IEC 10646-1:2000 UCS-4, respectively; the characters available in other non-  
20 default character types are not specified by this standard, except that one character in each nondefault  
21 character type shall be designated as a blank character to be used as a padding character.

#### 22 3.1.1 Letters

23 The twenty-six **letters** are:

24 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

25 The set of letters defines the syntactic class *letter*. The processor character set shall include lower-  
26 case and upper-case letters. A lower-case letter is equivalent to the corresponding upper-case letter in  
27 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)
```

#### 28 3.1.2 Digits

29 The ten **digits** are:

1      0 1 2 3 4 5 6 7 8 9

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

### 3    3.1.3    Underscore

4    R303    *underscore*                      is    \_

5    The underscore may be used as a significant character in a name.

### 6    3.1.4    Special characters

7    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

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

### 11    3.1.5    Other characters

12    Additional characters may be representable in the processor, but may appear only in comments (3.3.1.1,  
 13    3.3.2.1), character constants (4.4.5), input/output records (9.1.1), and character string edit descriptors  
 14    (10.3.2).

## 15    3.2    Low-level syntax

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

### 1 3.2.1 Names

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

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

5 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.

### 6 3.2.2 Constants

7 R305 *constant* **is** *literal-constant*

8 **or** *named-constant*

9 R306 *literal-constant* **is** *int-literal-constant*

10 **or** *real-literal-constant*

11 **or** *complex-literal-constant*

12 **or** *logical-literal-constant*

13 **or** *char-literal-constant*

14 **or** *boz-literal-constant*

15 R307 *named-constant* **is** *name*

16 R308 *int-constant* **is** *constant*

17 C302 (R308) *int-constant* shall be of type integer.

18 R309 *char-constant* **is** *constant*

19 C303 (R309) *char-constant* shall be of type character.

### 20 3.2.3 Operators

21 R310 *intrinsic-operator* **is** *power-op*

22 **or** *mult-op*

23 **or** *add-op*

24 **or** *concat-op*

25 **or** *rel-op*

26 **or** *not-op*

27 **or** *and-op*

28 **or** *or-op*

29 **or** *equiv-op*

30 R707 *power-op* **is** *\*\**

31 R708 *mult-op* **is** *\**

1		<b>or</b>	/
2	R709	<b>is</b>	+
3		<b>or</b>	-
4	R711	<b>is</b>	//
5	R713	<b>is</b>	.EQ.
6		<b>or</b>	.NE.
7		<b>or</b>	.LT.
8		<b>or</b>	.LE.
9		<b>or</b>	.GT.
10		<b>or</b>	.GE.
11		<b>or</b>	==
12		<b>or</b>	/=
13		<b>or</b>	<
14		<b>or</b>	<=
15		<b>or</b>	>
16		<b>or</b>	>=
17	R718	<b>is</b>	.NOT.
18	R719	<b>is</b>	.AND.
19	R720	<b>is</b>	.OR.
20	R721	<b>is</b>	.EQV.
21		<b>or</b>	.NEQV.
22		<b>or</b>	.XOR.
23	R311	<b>is</b>	<i>defined-unary-op</i>
24		<b>or</b>	<i>defined-binary-op</i>
25		<b>or</b>	<i>extended-intrinsic-op</i>
26	R703	<b>is</b>	. letter [ letter ] ... .
27	R723	<b>is</b>	. letter [ letter ] ... .
28	R312	<b>is</b>	<i>intrinsic-operator</i>

### 29 3.2.4 Statement labels

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

31 R313 *label* **is** *digit [ digit [ digit [ digit [ digit ] ] ] ]*

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

33 If a statement is labeled, the statement shall contain a nonblank character. The same statement label  
 34 shall not be given to more than one statement in a scoping unit. Leading zeros are not significant in  
 35 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.

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

- 2 (1) The label on a branch target statement (8.2) is used to identify that statement as the  
3 possible destination of a branch.
- 4 (2) The label on a FORMAT statement (10.2.1) is used to identify that statement as the format  
5 specification for a data transfer statement (9.5).
- 6 (3) In some forms of the DO construct (8.1.7), the range of the DO construct is identified by  
7 the label on the last statement in that range.

### 8 3.2.5 Delimiters

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

10 ( ... )  
11 / ... /  
12 [ ... ]  
13 (/ ... /)

## 14 3.3 Source form

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

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

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

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

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

### 24 3.3.1 Free source form

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

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

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

#### NOTE 3.5

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

```
REAL X
```

**NOTE 3.5 (cont.)**

```

READ 10
30 DO K=1,3

```

- 1 One or more blanks shall be used to separate adjacent keywords except in the following cases, where  
 2 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	

### 3 3.3.1.1 Free form commentary

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

**NOTE 3.6**

The standard does not restrict the number of consecutive comment lines.

### 10 3.3.1.2 Free form statement continuation

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

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

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



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

#### 5 **3.3.1.3 Free form statement termination**

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

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

#### 11 **3.3.1.4 Free form statements**

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

##### **NOTE 3.7**

No Fortran statement begins with a digit.

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

#### 14 **3.3.2 Fixed source form**

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

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

##### 19 **3.3.2.1 Fixed form commentary**

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

##### **NOTE 3.8**

The standard does not restrict the number of consecutive comment lines.

##### 26 **3.3.2.2 Fixed form statement continuation**

27 Except within commentary, character position 6 is used to indicate continuation. If character position 6 contains a blank  
28 or zero, the line is the initial line of a new statement, which begins in character position 7. If character position 6 contains  
29 any character other than blank or zero, character positions 7–72 of the line constitute a continuation of the preceding  
30 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.

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

### 1 3.3.2.3 Fixed form statement termination

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

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

### 8 3.3.2.4 Fixed form statements

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

## 14 3.4 Including source text

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

17 `INCLUDE char-literal-constant`

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

19 An INCLUDE line is not a Fortran statement.

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

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

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

31 The interpretation of *char-literal-constant* is processor dependent. An example of a possible valid inter-  
32 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;

**NOTE 3.10** (cont.)

- (4) For continued statements, place an ampersand (&) in both character position 73 of a continued line and character position 6 of a continuing line.

### 1 3.5 Macro processing

### 2 3.5.1 Macro definition

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

```

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

```

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

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

13 R316 *macro-attribute* **is** *access-spec*

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

[illegible]

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

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

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

27 C308 (R320) If *macro-expr* appears, when the macro is expanded it shall be of type integer, and have  
28 a non-negative value that specifies a representation method that exists on the processor.

29 A macro type declaration statement specifies that the named entities are macro local variables of the  
30 specified type. If the kind is not specified, they are of default kind. A macro local variable that is not a  
31 macro dummy argument shall appear in a macro type declaration statement.

```
32 R321  macro-body-block      is  [ macro-body-construct ] ...
```

```

33 R322  macro-body-construct      is macro-definition
34                                     or expand-stmt

```

			<b>or</b> <i>macro-body-stmt</i>
			<b>or</b> <i>macro-do-construct</i>
			<b>or</b> <i>macro-if-construct</i>
C309	A statement in a macro definition that is not a <i>macro-body-construct</i> or <i>macro-definition</i> shall not appear on a line with any other statement.		
R323	<i>macro-do-construct</i>	<b>is</b>	<i>macro-do-stmt</i> <i>macro-body-block</i> <i>macro-end-do-stmt</i>
R324	<i>macro-do-stmt</i>	<b>is</b>	MACRO DO <i>macro-do-variable-name</i> = <i>macro-do-limit</i> , ■ ■ <i>macro-do-limit</i> [ , <i>macro-do-limit</i> ]
C310	(R324) A <i>macro-do-variable-name</i> shall be a local variable of the macro being defined, and shall not be a macro dummy argument.		
R325	<i>macro-do-limit</i>	<b>is</b>	<i>macro-expr</i>
C311	(R325) A <i>macro-do-limit</i> shall expand to an expression of type integer.		
R326	<i>macro-end-do-stmt</i>	<b>is</b>	MACRO END DO
A macro DO construct iterates the expansion of its enclosed macro body block at macro expansion time. The number of iterations is determined by the values of the expanded macro expressions in the MACRO DO statement.			
R327	<i>macro-if-construct</i>	<b>is</b>	<i>macro-if-then-stmt</i> <i>macro-body-block</i> [ <i>macro-else-if-stmt</i> <i>macro-body-block</i> ] ... [ <i>macro-else-stmt</i> <i>macro-body-block</i> ] <i>macro-end-if-stmt</i>
R328	<i>macro-if-then-stmt</i>	<b>is</b>	MACRO IF ( <i>macro-condition</i> ) THEN
R329	<i>macro-else-if-stmt</i>	<b>is</b>	MACRO ELSE IF ( <i>macro-condition</i> ) THEN
R330	<i>macro-else-stmt</i>	<b>is</b>	MACRO ELSE
R331	<i>macro-end-if-stmt</i>	<b>is</b>	MACRO END IF
R332	<i>macro-condition</i>	<b>is</b>	<i>macro-expr</i>
C312	(R332) A macro condition shall expand to an expression of type logical.		
A macro IF construct provides conditional expansion of its enclosed macro body blocks at macro expansion time. Whether the enclosed macro body blocks contribute to the macro expansion is determined by the logical value of the expanded macro expressions in the MACRO IF and MACRO ELSE IF statements.			
R333	<i>macro-body-stmt</i>	<b>is</b>	<i>result-token</i> [ <i>result-token</i> ] ... [ && ]
C313	(R333) The first <i>result-token</i> shall not be MACRO unless the second <i>result-token</i> is not a keyword or name.		
R334	<i>result-token</i>	<b>is</b>	<i>token</i> [ %% <i>token</i> ] ...
C314	(R334) The concatenated textual <i>tokens</i> in a <i>result-token</i> shall have the form of a lexical token.		

- 1 R335 *token* **is** any lexical token including labels, keywords, and semi-colon.
- 2 C315 && shall not appear in the last *macro-body-stmt* of a macro definition.
- 3 C316 When a macro is expanded, the last *macro-body-stmt* processed shall not end with &&.
- 4 R336 *end-macro-stmt* **is** END MACRO [ *macro-name* ]
- 5 C317 (R314) The *macro-name* in the END MACRO statement shall be the same as the *macro-name*  
6 in the DEFINE MACRO statement.
- 7 R337 *macro-expr* **is** *basic-token-sequence*
- 8 C318 (R337) A *macro-expr* shall expand to a scalar initialization expression.
- 9 Macro expressions are used to control the behavior of the MACRO DO and MACRO IF constructs when  
10 a macro is being expanded. The type, type parameters, and value of a macro expression are determined  
11 when that macro expression is expanded.

## 12 3.5.2 Macro expansion

### 13 3.5.2.1 General

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

19 The process of macro expansion produces Fortran statements consisting of tokens. The combined length  
20 of the tokens for a single statement, plus inter-token spacing, shall not be greater than 33280 characters.  
21 If a statement contains any character that is not of default kind, the maximum number of characters  
22 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.

- 23 R338 *expand-stmt* **is** EXPAND *macro-name* [ ( *macro-actual-arg-list* ) ]
- 24 C319 (R338) *macro-name* shall be the name of a macro that was previously defined or accessed via  
25 use or host association.
- 26 C320 (R338) The macro shall expand to a sequence or zero or more complete Fortran statements.
- 27 C321 (R338) The statements produced by a macro expansion shall conform to the syntax rules and  
28 constraints as if they physically replaced the EXPAND statement prior to program processing.
- 29 C322 (R338) The statements produced by a macro expansion shall not include a statement which  
30 ends the scoping unit containing the EXPAND statement.
- 31 C323 (R338) If a macro expansion produces a statement which begins a new scoping unit, it shall also

- 1 produce a statement which ends that scoping unit.
- 2 C324 (R338) If the EXPAND statement appears as the *action-stmt* of an *if-stmt*, it shall expand to  
3 exactly one *action-stmt* that is not an *if-stmt*, *end-program-stmt*, *end-function-stmt*, or *end-*  
4 *subroutine-stmt*.
- 5 C325 (R338) If the EXPAND statement appears as a *do-term-action-stmt*, it shall expand to exactly one *action-stmt*  
6 that is not a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an  
7 *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.
- 8 C326 (R338) If the EXPAND statement has a label, the expansion of the macro shall produce at least  
9 one statement, and the first statement produced shall not have a label.
- 10 C327 (R338) A *macro-actual-arg* shall appear corresponding to each nonoptional macro dummy ar-  
11 gument.
- 12 C328 (R338) At most one *macro-actual-arg* shall appear corresponding to each optional macro dummy  
13 argument.
- 14 Expansion of a macro is performed by the EXPAND statement. If the EXPAND statement has a label,  
15 the label is interpreted after expansion as belonging to the first statement of the expansion.
- 16 R339 *macro-actual-arg* is [ *macro-dummy-name* = ] *macro-actual-arg-value*
- 17 C329 (R339) *macro-dummy-name* shall be the name of a macro dummy argument of the macro being  
18 expanded.
- 19 C330 (R338) The *macro-dummy-name*= shall not be omitted unless it has been omitted from each  
20 preceding *macro-actual-arg* in the *expand-stmt*.
- 21 R340 *macro-actual-arg-value* is basic-token-sequence
- 22 R341 *basic-token-sequence* is *basic-token*  
23 or [ *basic-token-sequence* ] *nested-token-sequence* ■  
24 ■ [ *basic-token-sequence* ]  
25 or *basic-token basic-token-sequence*
- 26 R342 *basic-token* is any lexical token except comma, parentheses, array ■  
27 ■ constructor delimiters, and semi-colon.
- 28 R343 *nested-token-sequence* is ( [ *arg-token* ] ... )  
29 or ( / [ *arg-token* ] ... / )  
30 or *lbracket* [ *arg-token* ] ... *rbracket*
- 31 R344 *arg-token* is *basic-token*  
32 or ,
- 33 Macro expansion processes any macro declarations of the macro definition, and then expands its macro  
34 body block. Any macro expressions in *macro-type-specs* are evaluated and the kinds of the macro  
35 variables thereby declared are determined for that particular expansion.
- 36 Macro expansion of a macro body block processes each macro body construct of the macro body block  
37 in turn, starting with the first macro body construct and ending with the last macro body construct.
- 38 Expansion of a statement within a macro body construct consists of three steps:
- 39 (1) token replacement,  
40 (2) token concatenation, and

1 (3) statement-dependent processing.

2 Token replacement replaces each token of a macro body statement or macro expression that is a macro  
 3 local variable with the value of that variable. In a macro expression, a reference to the PRESENT  
 4 intrinsic function with a macro dummy argument name as its actual argument is replaced by the token  
 5 .TRUE. if the specified macro dummy argument is present, and the token .FALSE. if the specified macro  
 6 dummy argument is not present. Otherwise, the value of a macro dummy argument that is present is  
 7 the sequence of tokens from the corresponding actual argument. The value of a macro dummy argument  
 8 that is not present is a zero-length token sequence. The value of an integer macro variable is its minimal-  
 9 length decimal representation; if negative this will produce two tokens, a minus sign and an unsigned  
 10 integer literal constant.

11 Token concatenation is performed with the %% operator, which is only permitted inside a macro defini-  
 12 tion. After expansion, each sequence of single tokens separated by %% operators is replaced by a single  
 13 token consisting of the concatenated text of the sequence of tokens. The result of a concatenation shall  
 14 be a valid Fortran token, and may be a different kind of token from one or more of the original sequence  
 15 of tokens.

#### NOTE 3.12

For example, the sequence

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

forms the single real literal constant 3.14159E+0.

### 16 3.5.2.2 Macro body statements

17 Processing a macro body statement produces a whole or partial Fortran statement. A macro body  
 18 statement that is either the first macro body statement processed by this macro expansion or the next  
 19 macro body statement processed after a macro body statement that did not end with the continuation  
 20 generation operator &&, is an initial macro body statement. The next macro body statement processed  
 21 after a macro body statement that ends with && is a continuation macro body statement. An initial  
 22 macro body statement that does not end with && produces a whole Fortran statement consisting of its  
 23 token sequence. All other macro body statements produce partial Fortran statements, and the sequence  
 24 of tokens starting with those produced by the initial macro body statement and appending the tokens  
 25 produced by each subsequent continuation macro body statement form a Fortran statement. The &&  
 26 operators are not included in the token sequence.

### 27 3.5.2.3 The macro DO construct

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

- 30 (1) The initial parameter  $m_1$ , the terminal parameter  $m_2$ , and the incrementation parameter  
 31  $m_3$  are of type integer with the same kind type parameter as the *macro-do-variable-name*.  
 32 Their values are given by the first *macro-expr*, the second *macro-expr*, and the third *macro-*  
 33 *expr* of the *macro-do-stmt* respectively, including, if necessary, conversion to the kind type  
 34 parameter of the *macro-do-variable-name* according to the rules for numeric conversion  
 35 (Table 7.11). If the third *macro-expr* does not appear,  $m_3$  has the value 1. The value of  $m_3$   
 36 shall not be zero.
- 37 (2) The macro DO variable becomes defined with the value of the initial parameter  $m_1$ .
- 38 (3) The **iteration count** is established and is the value of the expression  $(m_2 - m_1 + m_3)/m_3$ ,  
 39 unless that value is negative, in which case the iteration count is 0.

1 After this, the following steps are performed repeatedly until processing of the macro DO construct is  
 2 finished.

- 3 (1) The iteration count is tested. If it is zero, the loop terminates and processing of the macro  
 4 DO construct is finished.
- 5 (2) If the iteration count is nonzero, the macro body block of the macro DO construct is  
 6 expanded.
- 7 (3) The iteration count is decremented by one. The macro DO variable is incremented by the  
 8 value of the incrementation parameter  $m_3$ .

#### 9 3.5.2.4 The MACRO IF construct

10 The MACRO IF construct provides conditional expansion of macro body blocks. At most one of the  
 11 macro body blocks of the macro IF construct is expanded. The macro conditions of the construct are  
 12 evaluated in order until a true value is found or a MACRO ELSE or MACRO END IF statement is  
 13 encountered. If a true value or a MACRO ELSE statement is found, the macro body block immediately  
 14 following is expanded and this completes the processing of the construct. If none of the evaluated  
 15 conditions is true and there is no MACRO ELSE statement, the processing of the construct is completed  
 16 without expanding any of the macro body blocks within the construct.

#### 17 3.5.2.5 Macro definitions

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

#### 21 3.5.2.6 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)
  DO i=1,count
    EXPAND operation(i)
  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)
  ...
  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

```



## 1 4 Types

### 2 4.1 The concept of type

3 Fortran provides an abstract means whereby data may be categorized without relying on a particular  
4 physical representation. This abstract means is the concept of type.

5 A type has a name, a set of valid values, a means to denote such values (constants), and a set of  
6 operations to manipulate the values.

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

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

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

#### 11 4.1.1 Set of values

12 For each type, there is a set of valid values. The set of valid values may be completely determined,  
13 as is the case for logical and bits, or may be determined by a processor-dependent method, as is the  
14 case for integer, character, and real. For complex, the set of valid values consists of the set of all the  
15 combinations of the values of the individual components. For derived types, the set of valid values is as  
16 defined in 4.5.8.

#### 17 4.1.2 Constants

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

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

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

21 A structure constructor (4.5.10) may be used to construct a constant value of derived type from an  
22 appropriate sequence of initialization expressions (7.1.7). Such a constant value is considered to be a  
23 scalar even though the value may have components that are arrays.

#### 24 4.1.3 Operations

25 For each of the intrinsic types, a set of operations and corresponding operators is defined intrinsically.  
26 These are described in Clause 7. The intrinsic set may be augmented with operations and operators  
27 defined by functions with the OPERATOR interface (12.4.2.1). Operator definitions are described in  
28 Clauses 7 and 12.

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

### 31 4.2 Type parameters

32 A type may be parameterized. In this case, the set of values, the syntax for denoting the values, and  
33 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.
- 3 A kind type parameter may be used in initialization and specification expressions within the derived-type
- 4 definition (4.5.2) for the type; it participates in generic resolution (16.3.4). Each of the intrinsic types
- 5 has a kind type parameter named KIND, which is used to distinguish multiple representations of the
- 6 intrinsic type.

**NOTE 4.1**

By design, the value of a kind type parameter is 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. Generics are designed to be resolvable at compile time.

- 7 A length type parameter may be used in specification expressions within the derived-type definition for
- 8 the type, but it shall not be used in initialization expressions. The intrinsic character type has a length
- 9 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.

- 10 A type parameter value may be specified with a type specification (4.4, 4.5.9).

11 R401    *type-param-value*                      **is**   *scalar-int-expr*  
 12                                                      **or**   \*

13                                                      **or**   :

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

- 15 C402    (R401) A colon may be used as a *type-param-value* only in the declaration of an entity or
- 16           component that has the POINTER or ALLOCATABLE attribute.

- 17 A **deferred type parameter** is a length type parameter whose value can change during execution of
- 18 the program. A colon as a *type-param-value* specifies a deferred type parameter.

- 19 The values of the deferred type parameters of an object are determined by successful execution of an
- 20 ALLOCATE statement (6.3.1), execution of an intrinsic assignment statement (7.4.1.3), execution of a
- 21 pointer assignment statement (7.4.2), or by argument association (12.5.1.4).

**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.

- 22 An **assumed type parameter** is a length type parameter for a dummy argument that assumes the

- 1 type parameter value from the corresponding actual argument; it is also used for an associate name in a  
 2 SELECT TYPE construct that assumes the type parameter value from the corresponding selector. An  
 3 asterisk as a *type-param-value* specifies an assumed type parameter.

## 4 4.3 Relationship of types and values to objects

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

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

12 A variable is a data object. The type and type parameters of a variable determine which values that  
 13 variable may take. Assignment provides one means of defining or redefining the value of a variable of  
 14 any type. Assignment is defined intrinsically for all types where the type, type parameters, and shape  
 15 of both the variable and the value to be assigned to it are identical. Assignment between objects of  
 16 certain differing intrinsic types, type parameters, and shapes is described in Clause 7. A subroutine and  
 17 a generic interface (4.5.2, 12.4.2.1) whose generic specifier is ASSIGNMENT (=) define an assignment  
 18 that is not defined intrinsically or redefine an intrinsic derived-type assignment (7.4.1.4).

### NOTE 4.4

For example, assignment of a real value to an integer variable is defined intrinsically.

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

### 20 4.3.1 Type specifiers and type compatibility

#### 21 4.3.1.1 General

22 A type is specified by a **type specifier**. In an executable statement, or in an expression within a nonex-  
 23 ecutable statement, a *type-spec* is used. In a nonexecutable statement other than within an expression,  
 24 a *declaration-type-spec* is used.

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

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

28 R403 *declaration-type-spec* **is** *intrinsic-type-spec*  
 29 **or** TYPE ( *intrinsic-type-spec* )  
 30 **or** TYPE ( *derived-type-spec* )  
 31 **or** CLASS ( *derived-type-spec* )  
 32 **or** CLASS ( \* )

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

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

1 an extensible type.

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

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

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

#### NOTE 4.5

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

### 7 4.3.1.2 TYPE

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

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

### 16 4.3.1.3 CLASS

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

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

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

25 A nonpolymorphic entity is **type compatible** only with entities of the same declared type. A poly-  
26 morphic entity that is not an unlimited polymorphic entity is type compatible with entities of the same  
27 declared type or any of its extensions. Even though an unlimited polymorphic entity is not considered  
28 to have a declared type, it is type compatible with all entities. An entity is said to be type compatible  
29 with a type if it is type compatible with entities of that type.

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

31 A scalar entity of type bits is **bits compatible** with a scalar entity of type bits, integer, real, complex,  
32 or logical if and only if both entities have the same size expressed in bits.

#### J3 internal note

##### Unresolved Technical Issue 14

This might belong in the BITS subclause, or somewhere else. TKR compatibility belongs in c12 (the only place it is used), maybe this should go there? Or maybe split this subclause into one which handles type compatibilities of all kinds, and one which handles polymorphism? Anyway, it doesn't have anything to do with CLASS (title of subclause).

**J3 internal note**

## Unresolved Technical Issue 15

This leaves undefined whether entities of other types are bits compatible. Also, this term is often (always?) used as if it were symmetric, but like the other compatibility terms, it is not.

It has been suggested that

- (1) the definition is meant to be symmetric, and
- (2) complete.

Perhaps something like:

An entity is **bits compatible** with another entity if and only if one is of type bits, the other is of type bits, integer, real, complex, or logical, and scalar entities of these types have the same size expressed in bits.

It is not entirely obvious that this is correct, since the terms it is modelled on are not symmetric (and it is then used in those terms). The other terms are deliberately asymmetric in part to allow only safe associations, whereas it appears that a substantial point of BITS is perhaps to permit unsafe associations... if that is not the purpose of BITS, then making **bits compatible** symmetric is almost certainly incorrect.

**J3 internal note**

## Unresolved Technical Issue 16

It is unclear whether the term "bits compatible" is useful or necessary.

It is (mostly?) being used where otherwise we would be saying "type compatible with the same kind type parameters" (so-called TK compatibility). If that really is the only use, we should write the "TK compatibility" definition in one place and use it everywhere; otherwise, this extra complication is likely to lead to questions. (Alternative names for TK compatible are "kind compatible" and "type and kind compatible". The latter is probably the best.)

- 1 An entity is type, kind, and rank compatible, or **TKR compatible**, with another entity if both entities
- 2 have the same rank, and either the first entity is type compatible with the second and the kind type
- 3 parameters of the first entity have the same values as corresponding kind type parameters of the second,
- 4 or the two entities are bits compatible.

**J3 internal note**

## Unresolved Technical Issue 17

This has wrecked generic resolution. It allows ambiguous procedures to be added to a generic set. BAD BAD BAD. Fixable by making bits only match bits in a generic setting, i.e. disable bits compatibility the same way we already disable sequence association. Other fixes possible.

Note: If people want to use bits as a poor man's CLASS(\*), for purposes of describing some MPI routines in a shorter "clever" way, it is certainly not going to be straight-forward to fix this in any sort of consistent way.

Oh, and did I mention that the paragraph is getting harder to read?

**J3 internal note**

## Unresolved Technical Issue 18

Since the "kind" type parameter of bits is inherently a length type parameter (being the number of bits), it behooves us not to arbitrarily prevent variable length bits in the future. (FWIW, it does not look substantially more difficult to implement variable-length bits than fixed-length ones to me, and without adversely affecting the performance of known-length bit operations too.) This should be borne in mind while considering how to fix generics, for example.

**J3 internal note****Unresolved Technical Issue 19**

Given that having a “bits dummy” is semi-polymorphic, we should probably make “having a bits dummy argument” require an explicit interface, like we do with polymorphics.

Given the arguments (from the same people) that in order to avoid errors in co-arrays, one should have an explicit interface merely to *\*pass\** a co-indexed object as an actual argument, it is hard to avoid the conclusion that it is warranted in this case too.

Furthermore, requiring an explicit interface would facilitate future extension should that ever be contemplated.

- 1 A polymorphic allocatable object may be allocated to be of any type with which it is type compatible.
- 2 A polymorphic pointer or dummy argument may, during program execution, be associated with objects
- 3 with which it is type compatible.
- 4 The dynamic type of an allocated allocatable polymorphic object is the type with which it was allocated.
- 5 The dynamic type of an associated polymorphic pointer is the dynamic type of its target. The dynamic
- 6 type of a nonallocatable nonpointer polymorphic dummy argument is the dynamic type of its associated
- 7 actual argument. The dynamic type of an unallocated allocatable or a disassociated pointer is the same
- 8 as its declared type. The dynamic type of an entity identified by an associate name (8.1.5) is the dynamic
- 9 type of the selector with which it is associated. The dynamic type of an object that is not polymorphic
- 10 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).

**4.4 Intrinsic types****4.4.1 Classification and specification**

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

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

18	R404	<i>intrinsic-type-spec</i>	<b>is</b>	INTEGER [ <i>kind-selector</i> ]
19			<b>or</b>	REAL [ <i>kind-selector</i> ]
20			<b>or</b>	DOUBLE PRECISION
21			<b>or</b>	COMPLEX [ <i>kind-selector</i> ]
22			<b>or</b>	CHARACTER [ <i>char-selector</i> ]
23			<b>or</b>	LOGICAL [ <i>kind-selector</i> ]
24			<b>or</b>	BITS [ <i>kind-selector</i> ]
25	R405	<i>kind-selector</i>	<b>is</b>	( [ KIND = ] <i>scalar-int-initialization-expr</i> )

- 26 C408 (R405) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a rep-
- 27 resentation method that exists on the processor.

**4.4.2 Integer type**

- 29 The set of values for the **integer type** is a subset of the mathematical integers. The processor shall
- 30 provide one or more **representation methods** that define sets of values for data of type integer. Each



1 such method is characterized by a value for a type parameter called the **kind** type parameter; this kind  
 2 type parameter is of type default integer. The kind type parameter of a representation method is returned  
 3 by the intrinsic inquiry function KIND (13.7.96). The decimal exponent range of a representation method  
 4 is returned by the intrinsic function RANGE (13.7.143). The intrinsic function SELECTED\_INT\_KIND  
 5 (13.7.153) returns a kind value based on a specified decimal range requirement. The integer type includes  
 6 a zero value, which is considered to be neither negative nor positive. The value of a signed integer zero  
 7 is the same as the value of an unsigned integer zero.

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

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

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

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

14 R406	<i>signed-int-literal-constant</i>	<b>is</b>	[ <i>sign</i> ] <i>int-literal-constant</i>
15 R407	<i>int-literal-constant</i>	<b>is</b>	<i>digit-string</i> [ <i>_ kind-param</i> ]
16 R408	<i>kind-param</i>	<b>is</b>	<i>digit-string</i>
17		<b>or</b>	<i>scalar-int-constant-name</i>
18 R409	<i>signed-digit-string</i>	<b>is</b>	[ <i>sign</i> ] <i>digit-string</i>
19 R410	<i>digit-string</i>	<b>is</b>	<i>digit</i> [ <i>digit</i> ] ...
20 R411	<i>sign</i>	<b>is</b>	+
21		<b>or</b>	-

22 C409 (R408) A *scalar-int-constant-name* shall be a named constant of type integer.

23 C410 (R408) The value of *kind-param* shall be nonnegative.

24 C411 (R407) The value of *kind-param* shall specify a representation method that exists on the pro-  
 25 cessor.

26 The optional kind type parameter following *digit-string* specifies the kind type parameter of the integer  
 27 constant; if it is not present, the constant is of type default integer.

28 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.

### 29 4.4.3 Real type

30 The **real type** has values that approximate the mathematical real numbers. The processor shall provide  
 31 two or more **approximation methods** that define sets of values for data of type real. Each such method  
 32 has a **representation method** and is characterized by a value for a type parameter called the **kind**

1 type parameter; this kind type parameter is of type default integer. The kind type parameter of an  
 2 approximation method is returned by the intrinsic inquiry function KIND (13.7.96).

3 The decimal precision, decimal exponent range, and radix of an approximation method are returned by  
 4 the intrinsic functions PRECISION (13.7.137), RANGE (13.7.143), and RADIX (D13:RADIX (X)). The  
 5 intrinsic function SELECTED\_REAL\_KIND (13.7.154) returns a kind value based on specified precision,  
 6 range, and radix requirements.

#### NOTE 4.8

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

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

- 9 (1) in all relational operations,
- 10 (2) as actual arguments to intrinsic procedures other than those for which it is explicitly specified  
 11 that negative zero is distinguished, and
- 12 (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  $-0.0$ .

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

15 If the type keyword REAL is specified and the kind type parameter is not specified, the default kind  
 16 value is KIND (0.0) and the type specified is **default real**. If the type keyword DOUBLE PRECISION  
 17 is specified, the kind value is KIND (0.0D0) and the type specified is **double precision real**. The  
 18 decimal precision of the double precision real approximation method shall be greater than that of the  
 19 default real method.

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

- 23 R412 *signed-real-literal-constant* **is** [ *sign* ] *real-literal-constant*
- 24 R413 *real-literal-constant* **is** *significand* [ *exponent-letter exponent* ] [ *- kind-param* ]
- 25 **or** *digit-string exponent-letter exponent* [ *- kind-param* ]

1 R414 *significand* is *digit-string* . [ *digit-string* ]  
 2 or . *digit-string*  
 3 R415 *exponent-letter* is E  
 4 or D  
 5 R416 *exponent* is *signed-digit-string*

6 C412 (R413) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.

7 C413 (R413) The value of *kind-param* shall specify an approximation method that exists on the  
 8 processor.

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

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

15 The significand may be written with more digits than a processor will use to approximate the value of  
 16 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.

#### 17 4.4.4 Complex type

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

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

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

30 R417 *complex-literal-constant* is ( *real-part* , *imag-part* )  
 31 R418 *real-part* is *signed-int-literal-constant*

1 *or signed-real-literal-constant*  
 2 *or named-constant*  
 3 R419 *imag-part* *is signed-int-literal-constant*  
 4 *or signed-real-literal-constant*  
 5 *or named-constant*

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

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

13 If both the real and imaginary parts are integer, they are converted to the default real approximation  
 14 method and the constant is of type default complex. If only one of the parts is an integer, it is converted  
 15 to the approximation method selected for the part that is real and the kind type parameter value of the  
 16 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.

## 17 4.4.5 Character type

### 18 4.4.5.1 Character sets

19 The **character type** has a set of values composed of character strings. A **character string** is a sequence  
 20 of characters, numbered from left to right 1, 2, 3, ... up to the number of characters in the string. The  
 21 number of characters in the string is called the **length** of the string. The length is a type parameter; its  
 22 value is greater than or equal to zero. The length type parameter is of type default integer.

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

30 The character set defined by ISO/IEC 646:1991 (International Reference Version) is referred to as the  
 31 **ASCII character set** or the **ASCII character type**. The character set defined by ISO/IEC 10646-  
 32 1:2000 UCS-4 is referred to as the **ISO 10646 character set** or the **ISO 10646 character type**.

### 33 4.4.5.2 Character type specifier

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

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

```

3 R420  char-selector          is  length-selector
4                                     or  ( LEN = type-param-value , ■
5                                     ■ KIND = scalar-int-initialization-expr )
6                                     or  ( type-param-value , ■
7                                     ■ [ KIND = ] scalar-int-initialization-expr )
8                                     or  ( KIND = scalar-int-initialization-expr ■
9                                     ■ [ , LEN = type-param-value ] )
10 R421  length-selector       is  ( [ LEN = ] type-param-value )
11                                     or  * char-length [ , ]
12 R422  char-length           is  ( type-param-value )
13                                     or  scalar-int-literal-constant

```

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

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

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

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

- 19 (1) to declare a dummy argument,
- 20 (2) to declare a named constant,
- 21 (3) in the *type-spec* of an ALLOCATE statement wherein each *allocate-object* is a dummy
- 22 argument of type CHARACTER with an assumed character length,
- 23 (4) in the *type-spec* or *derived-type-spec* of a type guard statement (8.1.6), or
- 24 (5) in an external function, to declare the character length parameter of the function result.

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

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

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

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

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

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

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

- 43 (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.

#### 4.4.5.3 Character literal constant

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

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.

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

For the type character with kind *kind-param*, if present, and for type default character otherwise, a **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 standard 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.

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

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

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

The intrinsic operation **concatenation** (//) is defined between two data entities of type character (7.2.3)

- 1 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.

2 **4.4.5.4 Collating sequence**

- 3 The processor defines a collating sequence for the character set of each kind of character. A **collating**  
 4 **sequence** is a one-to-one mapping of the characters into the nonnegative integers such that each charac-  
 5 ter corresponds to a different nonnegative integer. The intrinsic functions CHAR (13.7.30) and ICHAR  
 6 (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.

- 7 The collating sequence of the default character type shall satisfy the following constraints.

- 8 (1) ICHAR ('A') < ICHAR ('B') < ... < ICHAR ('Z') for the twenty-six upper-case letters.  
 9 (2) ICHAR ('0') < ICHAR ('1') < ... < ICHAR ('9') for the ten digits.  
 10 (3) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('A') or  
 11 ICHAR (' ') < ICHAR ('A') < ICHAR ('Z') < ICHAR ('0').  
 12 (4) ICHAR ('a') < ICHAR ('b') < ... < ICHAR ('z') for the twenty-six lower-case letters.  
 13 (5) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('a') or  
 14 ICHAR (' ') < ICHAR ('a') < ICHAR ('z') < ICHAR ('0').

- 15 Except for blank, there are no constraints on the location of the special characters and underscore in  
 16 the collating sequence, nor is there any specified collating sequence relationship between the upper-case  
 17 and lower-case letters.

- 18 The collating sequence for the ASCII character type is as defined by ISO/IEC 646:1991 (International  
 19 Reference Version); this collating sequence is called the **ASCII collating sequence** in this standard.

- 1 The collating sequence for the ISO 10646 character type is as defined by ISO/IEC 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.

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

#### 5 **4.4.6 Logical type**

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

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

- 11 The type specifier for the logical type uses the keyword LOGICAL.

- 12 If the kind type parameter is not specified, the default kind value is KIND (.FALSE.) and the type  
13 specified is **default logical**.

- 14 R424 *logical-literal-constant*      **is** .TRUE. [ - *kind-param* ]  
15                                            **or** .FALSE. [ - *kind-param* ]

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

- 18 The optional kind type parameter following the trailing delimiter specifies the kind type parameter of  
19 the logical constant; if it is not present, the constant is of type default logical.

- 20 The intrinsic operations defined for data entities of logical type are: negation (.NOT.), conjunction  
21 (.AND.), inclusive disjunction (.OR.), logical equivalence (.EQV.), and logical nonequivalence (.NEQV.)  
22 as described in 7.2.5. There is also a set of intrinsically defined relational operators that compare the  
23 values of data entities of other types and yield a value of type default logical. These operations are  
24 described in 7.2.4.

#### 25 **4.4.7 Bits type**

##### 26 **4.4.7.1 General**

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

- 33 The type specifier for the bits type uses the keyword BITS.

- 34 If the kind type parameter is not specified for a bits variable, the default kind value is the size of a  
35 numeric storage unit expressed in bits, and the type specified is **default bits**.

- 36 R425 *boz-literal-constant*            **is** *binary-constant* [ - *kind-param* ]



```

1      or octal-constant [ - kind-param ]
2      or hex-constant [ - kind-param ]

```

3	R426	<i>binary-constant</i>	<b>is</b>	B ' <i>digit</i> [ <i>digit</i> ] ... '
4			<b>or</b>	B " <i>digit</i> [ <i>digit</i> ] ... "

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

6	R427	<i>octal-constant</i>	<b>is</b>	O ' <i>digit</i> [ <i>digit</i> ] ... '
7			<b>or</b>	O " <i>digit</i> [ <i>digit</i> ] ... "

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

9	R428	<i>hex-constant</i>	<b>is</b>	Z ' <i>hex-digit</i> [ <i>hex-digit</i> ] ... '
10			<b>or</b>	Z " <i>hex-digit</i> [ <i>hex-digit</i> ] ... "

11 R429 *hex-digit* is *digit*

12 **or** A

13 **or** B

14 or C

15 or D

16 or E

17 or F

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

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

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

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

**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.

## 1 4.5 Derived types

### 2 4.5.1 Derived type concepts

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

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

8 The **ultimate components** of an object of derived type are the components that are of intrinsic type  
9 or have the POINTER or ALLOCATABLE attribute, plus the ultimate components of the components  
10 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
```

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

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

### 16 4.5.2 Derived-type definition

#### 17 4.5.2.1 Syntax

18 R430	<i>derived-type-def</i>	<b>is</b>	<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>
24 R431	<i>derived-type-stmt</i>	<b>is</b>	TYPE [ [ , <i>type-attr-spec-list</i> ] :: ] <i>type-name</i> ■ ■ [ ( <i>type-param-name-list</i> ) ]
26 R432	<i>type-attr-spec</i>	<b>is</b>	ABSTRACT <b>or</b> <i>access-spec</i> <b>or</b> BIND (C) <b>or</b> EXTENDS ( <i>parent-type-name</i> )

30 C428 (R431) A derived type *type-name* shall not be DOUBLEPRECISION or the same as the name

- 1 of any intrinsic type defined in this part of ISO/IEC 1539.
- 2 C429 (R431) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.
- 3 C430 (R432) A *parent-type-name* shall be the name of a previously defined extensible type (4.5.7).
- 4 C431 (R430) If the type definition contains or inherits (4.5.7.2) a deferred binding (4.5.5), ABSTRACT  
5 shall appear.
- 6 C432 (R430) If ABSTRACT appears, the type shall be extensible.
- 7 C433 (R430) If EXTENDS appears, SEQUENCE shall not appear.
- 8 R433 *private-or-sequence*                    **is** *private-components-stmt*  
9                                                    **or** *sequence-stmt*
- 10 C434 (R430) The same *private-or-sequence* shall not appear more than once in a given *derived-type-*  
11 *def*.
- 12 R434 *end-type-stmt*                                                    **is** END TYPE [ *type-name* ]
- 13 C435 (R434) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in  
14 the corresponding *derived-type-stmt*.
- 15 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
```

#### 16 4.5.2.2 Accessibility

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

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

24 If a type definition is private, then the type name, and thus the structure constructor (4.5.10) for the  
25 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
```

**NOTE 4.21 (cont.)**

```

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.

**1 4.5.2.3 Sequence type**

2 R435 *sequence-stmt* **is** SEQUENCE

3 C436 (R439) If SEQUENCE appears, each data component shall be declared to be of an intrinsic type  
4 or of a sequence derived type.

5 C437 (R430) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.

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

**13 4.5.2.4 Determination of derived types**

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

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

**NOTE 4.25 (cont.)**

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 Declaration**

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 C438 (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 C439 (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 (7.4.1.3) 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 (7.1.6) in the *derived-type-*  
21 *def*. A kind type parameter may also be used as a primary in an initialization expression (7.1.7) 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)
```

**NOTE 4.27 (cont.)**

```
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 consists of the type parameter order of  
6 its parent type followed by any additional type parameters in the order of the type parameter list in the  
7 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 Syntax**

10	R439	<i>component-part</i>	<b>is</b>	[ <i>component-def-stmt</i> ] ...
11	R440	<i>component-def-stmt</i>	<b>is</b>	<i>data-component-def-stmt</i>
12			<b>or</b>	<i>proc-component-def-stmt</i>
13	R441	<i>data-component-def-stmt</i>	<b>is</b>	<i>declaration-type-spec</i> [ [ , <i>component-attr-spec-list</i> ] :: ] ■
14				■ <i>component-decl-list</i>
15	R442	<i>component-attr-spec</i>	<b>is</b>	<i>access-spec</i>
16			<b>or</b>	ALLOCATABLE
17			<b>or</b>	<i>component-dim-spec</i>
18			<b>or</b>	CONTIGUOUS
19			<b>or</b>	POINTER
20	R443	<i>component-decl</i>	<b>is</b>	<i>component-name</i> [ ( <i>component-array-spec</i> ) ] ■
21				■ [ <i>lbracket co-array-spec rbracket</i> ] ■
22				■ [ * <i>char-length</i> ] [ <i>component-initialization</i> ]

- 1 R444 *component-dim-spec*      **is** DIMENSION ( *component-array-spec* )  
 2      **or** DIMENSION [ ( *deferred-shape-spec-list* ) ] ■  
 3      ■ *lbracket co-array-spec rbracket*
- 4 R445 *component-array-spec*      **is** *explicit-shape-spec-list*  
 5      **or** *deferred-shape-spec-list*  
 6
- 7 C440 (R441) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.
- 8 C441 (R441) If neither the POINTER nor ALLOCATABLE attribute is specified, the *declaration-type-spec*  
 9 *spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived  
 10 type.
- 11 C442 (R441) If the POINTER or ALLOCATABLE attribute is specified, the *declaration-type-spec* in  
 12 the *component-def-stmt* shall be CLASS(\*) or shall specify an intrinsic type or any accessible  
 13 derived type including the type being defined.
- 14 C443 (R441) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec*  
 15 shall be a *deferred-shape-spec-list*.
- 16 C444 (R441) If a *co-array-spec* appears, it shall be a *deferred-shape-spec-list* and the component shall  
 17 have the ALLOCATABLE attribute.
- 18 C445 A data component whose type has a co-array ultimate component shall be a nonpointer nonal-  
 19 locatable scalar and shall not be a co-array.
- 20 C446 (R441) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each  
 21 *component-array-spec* shall be an *explicit-shape-spec-list*.
- 22 C447 (R445) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a  
 23 specification expression that does not contain references to specification functions or any object  
 24 designators other than named constants or subobjects thereof.
- 25 C448 (R441) A component shall not have both the ALLOCATABLE and the POINTER attribute.
- 26 C449 (R441) If the CONTIGUOUS attribute is specified, the component shall be an array with the  
 27 POINTER attribute.
- 28 C450 (R443) The \* *char-length* option is permitted only if the component is of type character.
- 29 C451 (R440) Each *type-param-value* within a *component-def-stmt* shall either be a colon, be an ini-  
 30 tialization expression, or be a specification expression that contains neither references to speci-  
 31 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*.

- 32 R446 *proc-component-def-stmt*      **is** PROCEDURE ( [ *proc-interface* ] ) , ■  
 33      ■ *proc-component-attr-spec-list* :: *proc-decl-list*

**NOTE 4.30**

See 12.4.2.3 for definitions of *proc-interface* and *proc-decl*.

- 34 R447 *proc-component-attr-spec*      **is** POINTER  
 35      **or** PASS [ ( *arg-name* ) ]



- 1                                   **or** NOPASS  
 2                                   **or** *access-spec*
- 3 C452   (R446) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-*  
 4           *component-def-stmt*.
- 5 C453   (R446) POINTER shall appear in each *proc-component-attr-spec-list*.
- 6 C454   (R446) If the procedure pointer component has an implicit interface or has no arguments,  
 7           NOPASS shall be specified.
- 8 C455   (R446) If PASS (*arg-name*) appears, the interface shall have a dummy argument named *arg-*  
 9           *name*.
- 10 C456   (R446) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.

#### 11 4.5.4.2 Array components

12 A data component is an array if its *component-decl* contains a *component-array-spec* or its *data-compo-*  
 13 *nent-def-stmt* contains the DIMENSION clause with a *component-array-spec*. If the *component-decl*  
 14 contains a *component-array-spec*, it specifies the array rank, and if the array is explicit shape (5.3.7.2),  
 15 the array bounds; otherwise, the *component-array-spec* in the DIMENSION clause specifies the array  
 16 rank, and if the array is explicit shape, the array bounds.

17 A data component is a co-array if its *component-decl* contains a *co-array-spec* or its *data-component-def-*  
 18 *stmt* contains a DIMENSION clause with a *co-array-spec*. If the *component-decl* contains a *co-array-spec*  
 19 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:

```

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

```

**NOTE 4.32 (cont.)**

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
  CHARACTER, DIMENSION (:), POINTER :: SYNOPSIS
END TYPE REFERENCE
```

Any object of type REFERENCE will have the four nonpointer components VOLUME, YEAR, PAGE, and TITLE, plus a pointer to an array of characters holding SYNOPSIS. The size of this 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 (7.4.2).

**4 4.5.4.4 The passed-object dummy argument**

- 5 A **passed-object dummy argument** is a distinguished dummy argument of a procedure pointer  
6 component or type-bound procedure. It affects procedure overriding (4.5.7.3) and argument association  
7 (12.5.1.2).
- 8 If NOPASS is specified, the procedure pointer component or type-bound procedure has no passed-object  
9 dummy argument.
- 10 If neither PASS nor NOPASS is specified or PASS is specified without *arg-name*, the first dummy argu-  
11 ment of a procedure pointer component or type-bound procedure is its passed-object dummy argument.
- 12 If PASS (*arg-name*) is specified, the dummy argument named *arg-name* is the passed-object dummy  
13 argument of the procedure pointer component or named type-bound procedure.
- 14 C457 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data  
15 object with the same declared type as the type being defined; all of its length type parameters  
16 shall be assumed; it shall be polymorphic (4.3.1.3) if and only if the type being defined is  
17 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.

#### 1 4.5.4.5 Default initialization for components

2 Default initialization provides a means of automatically initializing pointer components to be disasso-  
3 ciated, and nonpointer nonallocatable components to have a particular value. Allocatable components  
4 are always initialized to not allocated.

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

8	R448	<i>component-initialization</i>	<b>is</b> = <i>initialization-expr</i>
9			<b>or</b> => <i>null-init</i>
10			<b>or</b> => <i>initial-data-target</i>

```

11  R449  initial-data-target      is  designator

```

12 C458 (R441) If *component-initialization* appears, a double-colon separator shall appear before the  
13 *component-decl-list*.

14 C459 (R441) If *component-initialization* appears, every type parameter and array bound of the com-  
15 ponent shall be an initialization expression.

16 C460 (R441) If => appears in *component-initialization*, POINTER shall appear in the *component-*  
17 *attr-spec-list*. If = appears in *component-initialization*, POINTER or ALLOCATABLE shall  
18 not appear in the *component-attr-spec-list*.

19 C461 (R448) If *initial-data-target* appears, *component-name* shall be data-pointer-initialization com-  
20 patible with it.

21 C462 (R449) The *designator* shall designate a variable that is an initialization target. Every subscript,  
22 section subscript, substring starting point, and substring ending point in *designator* shall be an  
23 initialization expression.

24 If *null-init* appears for a pointer component, that component in any object of the type has an initial  
25 association status of disassociated (16.5.2.2) or becomes disassociated as specified in 16.5.2.2.2.

26 A variable is an **initialization target** if it has the TARGET attribute, either has the SAVE attribute  
27 or is declared in the main program, does not have the ALLOCATABLE attribute.

28 If *initial-data-target* appears for a data pointer component, that component in any object of the type is  
29 initially associated with the target or becomes associated with the target as specified in 16.5.2.2.1.

30 If *initial-proc-target* appears in *proc-decl* for a procedure pointer component, that component in any  
31 object of the type is initially associated with the target or becomes associated with the target as specified  
32 in 16.5.2.2.1.

33 If *initialization-expr* appears for a nonpointer component, that component in any object of the type  
34 is initially defined (16.6.3) or becomes defined as specified in 16.6.5 with the value determined from  
35 *initialization-expr*. If necessary, the value is converted according to the rules of intrinsic assignment  
36 (7.4.1.3) to a value that agrees in type, type parameters, and shape with the component. If the compo-  
37 nent is of a type for which default initialization is specified for a component, the default initialization  
38 specified by *initialization-expr* overrides the default initialization specified for that component. When  
39 one initialization **overrides** another it is as if only the overriding initialization were specified (see Note

- 1 4.37). Explicit initialization in a type declaration statement (5.2) overrides default initialization (see
- 2 Note 4.36). Unlike explicit initialization, default initialization does not imply that the object has the
- 3 SAVE attribute.
- 4 A subcomponent (6.1.2) is **default-initialized** if the type of the object of which it is a component
- 5 specifies default initialization for that component, and the subcomponent is not a subobject of an object
- 6 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
  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.

**NOTE 4.38 (cont.)**

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

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 R450 *private-components-stmt* is PRIVATE

11 C463 (R450) A *private-components-stmt* is permitted only if the type definition is within the specifi-  
12 cation part of a module.

- 1 The default accessibility for the components that are declared in a type's *component-part* is private
- 2 if the type definition contains a *private-components-stmt*, and public otherwise. The accessibility of a
- 3 component may be explicitly declared by an *access-spec*; otherwise its accessibility is the default for the
- 4 type definition in which it is declared.
- 5 If a component is private, that component name is accessible only within the module containing the
- 6 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
```

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.

### 7 4.5.5 Type-bound procedures

- 8 R451 *type-bound-procedure-part* **is** *contains-stmt*
- 9 [ *binding-private-stmt* ]
- 10 [ *proc-binding-stmt* ] ...
- 11 R452 *binding-private-stmt* **is** PRIVATE
- 12 C464 (R451) A *binding-private-stmt* is permitted only if the type definition is within the specification

- 1 part of a module.
- 2 R453 *proc-binding-stmt*           **is** *specific-binding*  
3                                   **or** *generic-binding*  
4                                   **or** *final-binding*
- 5 R454 *specific-binding*           **is** PROCEDURE [ ( *interface-name* ) ] ■  
6                                   ■ [ [ , *binding-attr-list* ] :: ] ■  
7                                   ■ *binding-name* [ => *procedure-name* ]
- 8 C465 (R454) If => *procedure-name* appears, the double-colon separator shall appear.
- 9 C466 (R454) If => *procedure-name* appears, *interface-name* shall not appear.
- 10 C467 (R454) The *procedure-name* shall be the name of an accessible module procedure or an external  
11 procedure that has an explicit interface.
- 12 If neither => *procedure-name* nor *interface-name* appears, it is as though => *procedure-name* had  
13 appeared with a procedure name the same as the binding name.
- 14 R455 *generic-binding*           **is** GENERIC ■  
15                                   ■ [ , *access-spec* ] :: *generic-spec* => *binding-name-list*
- 16 C468 (R455) Within the *specification-part* of a module, each *generic-binding* shall specify, either  
17 implicitly or explicitly, the same accessibility as every other *generic-binding* with that *generic-*  
18 *spec* in the same derived type.
- 19 C469 (R455) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the  
20 type.
- 21 C470 (R455) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object  
22 dummy argument (4.5.4.4).
- 23 C471 (R455) If *generic-spec* is OPERATOR ( *defined-operator* ), the interface of each binding shall  
24 be as specified in 12.4.2.1.1.
- 25 C472 (R455) If *generic-spec* is ASSIGNMENT ( = ), the interface of each binding shall be as specified  
26 in 12.4.2.1.2.
- 27 C473 (R455) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in  
28 9.5.3.7. The type of the *dtv* argument shall be *type-name*.
- 29 R456 *binding-attr*               **is** PASS [ ( *arg-name* ) ]  
30                                   **or** NOPASS  
31                                   **or** NON\_OVERRIDABLE  
32                                   **or** DEFERRED  
33                                   **or** *access-spec*
- 34 C474 (R456) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.
- 35 C475 (R454) If the interface of the binding has no dummy argument of the type being defined,  
36 NOPASS shall appear.
- 37 C476 (R454) If PASS ( *arg-name* ) appears, the interface of the binding shall have a dummy argument  
38 named *arg-name*.
- 39 C477 (R456) PASS and NOPASS shall not both appear in the same *binding-attr-list*.
- 40 C478 (R456) NON\_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-*

1 *attr-list*.

2 C479 (R456) DEFERRED shall appear if and only if *interface-name* appears.

3 C480 (R454) An overriding binding (4.5.7.3) shall have the DEFERRED attribute only if the binding  
4 it overrides is deferred.

5 C481 (R454) A binding shall not override an inherited binding (4.5.7.2) that has the NON\_OVER-  
6 RIDABLE attribute.

7 Each binding in a *proc-binding-stmt* specifies a **type-bound procedure**. A type-bound procedure  
8 may have a passed-object dummy argument (4.5.4.4). A *generic-binding* specifies a type-bound generic  
9 interface for its specific bindings. A binding that specifies the DEFERRED attribute is a **deferred**  
10 **binding**. A deferred binding shall appear only in the definition of an abstract type.

11 A type-bound procedure may be identified by a **binding name** in the scope of the type definition.  
12 This name is the *binding-name* for a specific binding, and the *generic-name* for a generic binding whose  
13 *generic-spec* is *generic-name*. A final binding, or a generic binding whose *generic-spec* is not *generic-*  
14 *name*, has no binding name.

15 The interface of a specific binding is that of the procedure specified by *procedure-name* or the interface  
16 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
```

17 The same *generic-spec* may be used in several *generic-bindings* within a single derived-type definition.

18 Each additional *generic-binding* 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).

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

23 A public type-bound procedure is accessible via any accessible object of the type. A private type-bound  
24 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*.

1 **4.5.6 Final subroutines**2 **4.5.6.1 Declaration**

3 R457 *final-binding*                                    **is** FINAL [ :: ] *final-subroutine-name-list*

4 C482 (R457) A *final-subroutine-name* shall be the name of a module procedure with exactly one  
 5 dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocat-  
 6 able, nonpolymorphic variable of the derived type being defined. All length type parameters of  
 7 the dummy argument shall be assumed. The dummy argument shall not be `INTENT(OUT)`.

8 C483 (R457) A *final-subroutine-name* shall not be one previously specified as a final subroutine for  
 9 that type.

10 C484 (R457) A final subroutine shall not have a dummy argument with the same kind type parameters  
 11 and rank as the dummy argument of another final subroutine of that type.

12 The `FINAL` keyword specifies a list of **final subroutines**. A final subroutine might be executed when  
 13 a data entity of that type is finalized (4.5.6.2).

14 A derived type is **finalizable** if it has any final subroutines or if it has any nonpointer, nonallocatable  
 15 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.

16 **4.5.6.2 The finalization process**

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

- 19 (1) If the dynamic type of the entity has a final subroutine whose dummy argument has the  
 20 same kind type parameters and rank as the entity being finalized, it is called with the entity  
 21 as an actual argument. Otherwise, if there is an elemental final subroutine whose dummy  
 22 argument has the same kind type parameters as the entity being finalized, it is called with  
 23 the entity as an actual argument. Otherwise, no subroutine is called at this point.
- 24 (2) Each finalizable component that appears in the type definition is finalized. If the entity  
 25 being finalized is an array, each finalizable component of each element of that entity is  
 26 finalized separately.
- 27 (3) If the entity is of extended type and the parent type is finalizable, the parent component is  
 28 finalized.

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

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

#### 5 **4.5.6.3 When finalization occurs**

6 When a pointer is deallocated its target is finalized. When an allocatable entity is deallocated, it is  
7 finalized.

8 A nonpointer, nonallocatable object that is not a dummy argument or function result is finalized immedi-  
9 ately before it would become undefined due to execution of a RETURN or END statement (16.6.6, item  
10 (3)). If the object is defined in a module or submodule, and there are no longer any active procedures  
11 referencing the module or submodule, it is processor-dependent whether it is finalized.

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

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

16 If an executable construct references a structure constructor or array constructor, the entity created by  
17 the constructor is finalized after execution of the innermost executable construct containing the reference.

18 If a specification expression in a scoping unit references a function, the result is finalized before execution  
19 of the executable constructs in the scoping unit.

20 If a specification expression in a scoping unit references a structure constructor or array constructor, the  
21 entity created by the constructor is finalized before execution of the executable constructs in the scoping  
22 unit.

23 When a procedure is invoked, a nonpointer, nonallocatable object that is an actual argument associated  
24 with an INTENT(OUT) dummy argument is finalized.

25 When an intrinsic assignment statement is executed, the variable is finalized after evaluation of *expr*  
26 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 assign-  
ment.

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

#### 30 **4.5.6.4 Entities that are not finalized**

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

#### **NOTE 4.52**

A nonpointer, nonallocatable object that has the SAVE attribute or that occurs in the main pro-  
gram is never finalized as a direct consequence of the execution of a RETURN or END statement.

**NOTE 4.52 (cont.)**

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.

1 **4.5.7 Type extension**2 **4.5.7.1 Concepts**

3 A nonsequence derived type that does not have the BIND attribute is an **extensible type**.

4 An extensible type that does not have the EXTENDS attribute is a **base type**. A type that has the  
 5 EXTENDS attribute is an **extended type**. The **parent type** of an extended type is the type named  
 6 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.

7 A base type is an **extension type** of itself only. An extended type is an extension of itself and of all  
 8 types for which its parent type is an extension.

9 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.

10 **4.5.7.2 Inheritance**

11 An extended type includes all of the type parameters, all of the components, and the nonoverridden  
 12 (4.5.7.3) nonfinal procedure bindings of its parent type. These are said to be **inherited** by the extended  
 13 type from the parent type. They retain all of the attributes that they had in the parent type. Additional  
 14 type parameters, components, and procedure bindings may be declared in the derived-type definition of  
 15 the 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.

- 1 An extended type has a scalar, nonpointer, nonallocatable, **parent component** with the type and  
 2 type parameters of the parent type. The name of this component is the parent type name. It has the  
 3 accessibility of the parent type. Components of the parent component are **inheritance associated**  
 4 (16.5.4) with the corresponding components inherited from the parent type. An **ancestor component**  
 5 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
END TYPE COLOR_POINT

```

**6 4.5.7.3 Type-bound procedure overriding**

- 7 If a nongeneric binding specified in a type definition has the same binding name as a binding from the  
 8 parent type then the binding specified in the type definition **overrides** the one from the parent type.
- 9 The overriding binding and the overridden binding shall satisfy the following conditions.
- 10 (1) Either both shall have a passed-object dummy argument or neither shall.
  - 11 (2) If the overridden binding is pure then the overriding binding shall also be pure.
  - 12 (3) Either both shall be elemental or neither shall.
  - 13 (4) They shall have the same number of dummy arguments.
  - 14 (5) Passed-object dummy arguments, if any, shall correspond by name and position.
  - 15 (6) Dummy arguments that correspond by position shall have the same names and characteris-  
 16 tics, except for the type of the passed-object dummy arguments.
  - 17 (7) Either both shall be subroutines or both shall be functions having the same result charac-  
 18 teristics (12.3.2).
  - 19 (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:

**NOTE 4.59 (cont.)**

```

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

```

- 1 If a generic binding specified in a type definition has the same *generic-spec* as an inherited binding, it
- 2 extends the generic interface and shall satisfy the requirements specified in 16.3.4.
- 3 If a generic binding in a type definition has the same *dtio-generic-spec* as one inherited from the parent, it
- 4 extends the type-bound generic interface for *dtio-generic-spec* and shall satisfy the requirements specified
- 5 in 16.3.4.
- 6 A binding of a type and a binding of an extension of that type are said to correspond if the latter binding
- 7 is the same binding as the former, overrides a corresponding binding, or is an inherited corresponding
- 8 binding.

**9 4.5.8 Derived-type values**

10 The **component value** of

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

15 The set of values of a particular derived type consists of all possible sequences of the component values

16 of its components.

**17 4.5.9 Derived-type specifier**

18 A derived-type specifier is used in several contexts to specify a particular derived type and type param-

19 eters.

20 R458 *derived-type-spec*                    **is** *type-name* [ ( *type-param-spec-list* ) ]

21 R459 *type-param-spec*                    **is** [ *keyword* = ] *type-param-value*

22 C485 (R458) *type-name* shall be the name of an accessible derived type.

23 C486 (R458) *type-param-spec-list* shall appear only if the type is parameterized.

24 C487 (R458) There shall be at most one *type-param-spec* corresponding to each parameter of the type.

25 If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding

26 to that type parameter.

27 C488 (R459) The *keyword*= may be omitted from a *type-param-spec* only if the *keyword*= has been

- 1 omitted from each preceding *type-param-spec* in the *type-param-spec-list*.
- 2 C489 (R459) Each *keyword* shall be the name of a parameter of the type.
- 3 C490 (R459) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the decla-  
4 ration of a dummy argument or associate name or in the allocation of a dummy argument.
- 5 Type parameter values that do not have type parameter keywords specified correspond to type param-  
6 eters in type parameter order (4.5.3.2). If a type parameter keyword is present, the value is assigned to  
7 the type parameter named by the keyword. If necessary, the value is converted according to the rules of  
8 intrinsic assignment (7.4.1.3) to a value of the same kind as the type parameter.
- 9 The value of a type parameter for which no *type-param-value* has been specified is its default value.

#### 10 4.5.10 Construction of derived-type values

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

14 R460	<i>structure-constructor</i>	<b>is</b>	<i>derived-type-spec</i> ( [ <i>component-spec-list</i> ] )
15 R461	<i>component-spec</i>	<b>is</b>	[ <i>keyword</i> = ] <i>component-data-source</i>
16 R462	<i>component-data-source</i>	<b>is</b>	<i>expr</i>
17		<b>or</b>	<i>data-target</i>
18		<b>or</b>	<i>proc-target</i>

- 19 C491 (R460) The *derived-type-spec* shall not specify an abstract type (4.5.7).
- 20 C492 (R460) At most one *component-spec* shall be provided for a component.
- 21 C493 (R460) If a *component-spec* is provided for an ancestor component, a *component-spec* shall not  
22 be provided for any component that is inheritance associated with a subcomponent of that  
23 ancestor component.
- 24 C494 (R460) A *component-spec* shall be provided for a nonallocatable component unless it has default  
25 initialization or is inheritance associated with a subcomponent of another component for which  
26 a *component-spec* is provided.
- 27 C495 (R461) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been  
28 omitted from each preceding *component-spec* in the constructor.
- 29 C496 (R461) Each *keyword* shall be the name of a component of the type.
- 30 C497 (R460) The type name and all components of the type for which a *component-spec* appears shall  
31 be accessible in the scoping unit containing the structure constructor.
- 32 C498 (R460) If *derived-type-spec* is a type name that is the same as a generic name, the *component-*  
33 *spec-list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a  
34 generic reference (12.5.4.1).
- 35 C499 (R462) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall  
36 correspond to a procedure pointer component.
- 37 C4100 (R462) 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*.

- 1 In the absence of a component keyword, each *component-data-source* is assigned to the corresponding
- 2 component in component order (4.5.4.6). If a component keyword appears, the *expr* is assigned to the
- 3 component named by the keyword. For a nonpointer component, the declared type and type parameters
- 4 of the component and *expr* shall conform in the same way as for a *variable* and *expr* in an intrinsic
- 5 assignment statement (7.4.1.2), as specified in Table 7.10. If necessary, each value of intrinsic type is
- 6 converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type and type
- 7 parameters with the corresponding component of the derived type. For a nonpointer nonallocatable
- 8 component, the shape of the expression shall conform with the shape of the component.
- 9 If a component with default initialization has no corresponding *component-data-source*, then the default
- 10 initialization is applied to that component. If an allocatable component has no corresponding *component-*
- 11 *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 may 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)
                                           ! has private components
COLOR_POINT( 1, 2, 3)                     ! All components of TYPE(point)
                                           ! need to be accessible.
```

- 12 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. /) )
```

- 13 For a pointer component, the corresponding *component-data-source* shall be an allowable *data-target* or
- 14 *proc-target* for such a pointer in a pointer assignment statement (7.4.2). If the component data source is

- 1 a pointer, the association of the component is that of the pointer; otherwise, the component is pointer  
 2 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.

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

## 12 4.5.11 Derived-type operations and assignment

- 13 Intrinsic assignment of derived-type entities is described in 7.4.1. This part of ISO/IEC 1539 does  
 14 not specify any intrinsic operations on derived-type entities. Any operation on derived-type entities  
 15 or defined assignment (7.4.1.4) for derived-type entities shall be defined explicitly by a function or a  
 16 subroutine, and a generic interface (4.5.2, 12.4.2.1).

## 17 4.6 Enumerations and enumerators

- 18 An enumeration is a set of enumerators. An enumerator is a named integer constant. An enumeration  
 19 definition specifies the enumeration and its set of enumerators of the corresponding integer kind.

- |         |                            |           |                                          |
|---------|----------------------------|-----------|------------------------------------------|
| 20 R463 | <i>enum-def</i>            | <b>is</b> | <i>enum-def-stmt</i>                     |
| 21      |                            |           | <i>enumerator-def-stmt</i>               |
| 22      |                            |           | [ <i>enumerator-def-stmt</i> ] ...       |
| 23      |                            |           | <i>end-enum-stmt</i>                     |
| 24 R464 | <i>enum-def-stmt</i>       | <b>is</b> | ENUM, BIND(C)                            |
| 25 R465 | <i>enumerator-def-stmt</i> | <b>is</b> | ENUMERATOR [ :: ] <i>enumerator-list</i> |



- 1 R466 *enumerator* **is** *named-constant* [ = *scalar-int-initialization-expr* ]  
 2 R467 *end-enum-stmt* **is** END ENUM
- 3 C4101 (R465) If = appears in an *enumerator*, a double-colon separator shall appear before the *enu-*  
 4 *merator-list*.
- 5 For an enumeration, the kind is selected such that an integer type with that kind is interoperable (15.3.2)  
 6 with the corresponding C enumeration type. The corresponding C enumeration type is the type that  
 7 would be declared by a C enumeration specifier (6.7.2.2 of the C International Standard) that specified  
 8 C enumeration constants with the same values as those specified by the *enum-def*, in the same order as  
 9 specified by the *enum-def*.
- 10 The companion processor (2.5.10) shall be one that uses the same representation for the types declared  
 11 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 values of any such enumerators will be interoperable with the values 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 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.

- 12 An enumerator is treated as if it were explicitly declared with the PARAMETER attribute. The enu-  
 13 merator is defined in accordance with the rules of intrinsic assignment (7.4) with the value determined  
 14 as follows.
- 15 (1) If *scalar-int-initialization-expr* is specified, the value of the enumerator is the result of  
 16 *scalar-int-initialization-expr*.
- 17 (2) If *scalar-int-initialization-expr* is not specified and the enumerator is the first enumerator  
 18 in *enum-def*, the enumerator has the value 0.
- 19 (3) If *scalar-int-initialization-expr* is not specified and the enumerator is not the first enumer-  
 20 ator in *enum-def*, its value is the result of adding 1 to the value of the enumerator that  
 21 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
```

**NOTE 4.69 (cont.)**

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.

## 1 4.7 Construction of array values

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

4	R468	<i>array-constructor</i>	<b>is</b>	(/ <i>ac-spec</i> /)
5			<b>or</b>	<i>lbracket ac-spec rbracket</i>
6	R469	<i>ac-spec</i>	<b>is</b>	<i>type-spec</i> ::
7			<b>or</b>	[ <i>type-spec</i> ::] <i>ac-value-list</i>
8	R470	<i>lbracket</i>	<b>is</b>	[
9	R471	<i>rbracket</i>	<b>is</b>	]
10	R472	<i>ac-value</i>	<b>is</b>	<i>expr</i>
11			<b>or</b>	<i>ac-implied-do</i>
12	R473	<i>ac-implied-do</i>	<b>is</b>	( <i>ac-value-list</i> , <i>ac-implied-do-control</i> )
13	R474	<i>ac-implied-do-control</i>	<b>is</b>	<i>ac-do-variable</i> = <i>scalar-int-expr</i> , <i>scalar-int-expr</i> ■
14				■ [ , <i>scalar-int-expr</i> ]
15	R475	<i>ac-do-variable</i>	<b>is</b>	<i>do-variable</i>
16	C4102	(R469) If <i>type-spec</i> is omitted, each <i>ac-value</i> expression in the <i>array-constructor</i> shall have the		
17		same type and kind type parameters.		
18	C4103	(R469) If <i>type-spec</i> specifies an intrinsic type, each <i>ac-value</i> expression in the <i>array-constructor</i>		
19		shall be of an intrinsic type that is in type conformance with a variable of type <i>type-spec</i> as		
20		specified in Table 7.10.		
21	C4104	(R469) If <i>type-spec</i> specifies a derived type, all <i>ac-value</i> expressions in the <i>array-constructor</i>		
22		shall be of that derived type and shall have the same kind type parameter values as specified by		
23		<i>type-spec</i> .		
24	C4105	(R473) The <i>ac-do-variable</i> of an <i>ac-implied-do</i> that is in another <i>ac-implied-do</i> shall not appear		
25		as the <i>ac-do-variable</i> of the containing <i>ac-implied-do</i> .		

- 26 If *type-spec* is omitted, each *ac-value* expression in the array constructor shall have the same length type  
 27 parameters; in this case, the type and type parameters of the array constructor are those of the *ac-value*  
 28 expressions.

- 1 If *type-spec* appears, it specifies the type and type parameters of the array constructor. Each *ac-value*
- 2 expression in the *array-constructor* shall be compatible with intrinsic assignment to a variable of this
- 3 type and type parameters. Each value is converted to the type parameters of the *array-constructor* in
- 4 accordance with the rules of intrinsic assignment (7.4.1.3).
- 5 The character length of an *ac-value* in an *ac-implied-do* whose iteration count is zero shall not depend
- 6 on the value of the *ac-do-variable* and shall not depend on the value of an expression that is not an
- 7 initialization expression.
- 8 If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-*
- 9 *value* is an array expression, the values of the elements of the expression, in array element order (6.2.2.2),
- 10 specify the corresponding sequence of elements of the array constructor. If an *ac-value* is an *ac-implied-*
- 11 *do*, it is expanded to form a sequence of elements under the control of the *ac-do-variable*, as in the DO
- 12 construct (8.1.7.5).
- 13 For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct.
- 14 An empty sequence forms a zero-sized rank-one 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  $3 \times 2$  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)
```

**NOTE 4.74 (cont.)**

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 the constants with the same character length.

## 1 5 Attribute declarations and specifications

### 2 5.1 General

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

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

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

### 12 5.2 Type declaration statements

#### 13 5.2.1 Syntax

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

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

18 R502 *attr-spec*                            **is** *access-spec*  
 19                                            **or** ALLOCATABLE  
 20                                            **or** ASYNCHRONOUS  
 21                                            **or** CONTIGUOUS  
 22                                            **or** dimension-spec  
 23                                            **or** EXTERNAL  
 24                                            **or** INTENT ( *intent-spec* )  
 25                                            **or** INTRINSIC  
 26                                            **or** *language-binding-spec*  
 27                                            **or** OPTIONAL  
 28                                            **or** PARAMETER  
 29                                            **or** POINTER  
 30                                            **or** PROTECTED  
 31                                            **or** SAVE  
 32                                            **or** TARGET  
 33                                            **or** VALUE  
 34                                            **or** VOLATILE  
 35

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

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

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

#### NOTE 5.1

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

36 If a type parameter in a *declaration-type-spec* or in a *char-length* in an *entity-decl* is defined by an  
37 expression that is not an initialization expression, the type parameter value is established on entry to  
38 the procedure and is not affected by any redefinition or undefinition of the variables in the expression

1 during execution of the procedure.

## 2 5.2.2 Initialization

3 The appearance of *initialization* in an *entity-decl* for an entity without the PARAMETER attribute  
 4 specifies that the entity is a variable with **explicit initialization**. Explicit initialization alternatively  
 5 may be specified in a DATA statement unless the variable is of a derived type for which default initial-  
 6 ization is specified. If *initialization* is *=initialization-expr*, the variable is initially defined with the value  
 7 specified by the *initialization-expr*; if necessary, the value is converted according to the rules of intrinsic  
 8 assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the variable. A  
 9 variable, or part of a variable, shall not be explicitly initialized more than once in a program. If the  
 10 variable is an array, it shall have its shape specified in either the type declaration statement or a previous  
 11 attribute specification statement in the same scoping unit.

12 If *null-init* appears, the initial association status of the object is disassociated. If *initial-data-target*  
 13 appears, the object is initially associated with the target.

14 Explicit initialization of a variable that is not in a common block implies the SAVE attribute, which  
 15 may be confirmed by explicit specification.

## 16 5.2.3 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.)

## 17 5.3 Attributes

### 18 5.3.1 Constraints

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

21 C513 An entity shall not be explicitly given any attribute more than once in a scoping unit.

22 C514 An *array-spec* for a function result that does not have the ALLOCATABLE or POINTER  
 23 attribute shall be an *explicit-shape-spec-list*.

24 C515 The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy

1 argument of a procedure that has a *proc-language-binding-spec*.

## 2 5.3.2 Accessibility attribute

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

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

6 C516 (R507) An *access-spec* shall appear only in the *specification-part* of a module.

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

### NOTE 5.4

An example of an accessibility specification is:

REAL, PRIVATE :: X, Y, Z

## 13 5.3.3 ALLOCATABLE attribute

14 An entity with the **ALLOCATABLE attribute** is a variable for which space is allocated by an AL-  
15 LOCATE statement (6.3.1) or by an intrinsic assignment statement (7.4.1.3).

## 16 5.3.4 ASYNCHRONOUS attribute

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

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

- 20 (1) the variable appears in an executable statement or specification expression in that scoping  
21 unit and
- 22 (2) any statement of the scoping unit is executed while the variable is a pending I/O storage  
23 sequence affector (9.5.1.4).

24 Use of a variable in an asynchronous input/output statement can imply the ASYNCHRONOUS attribute;  
25 see subclause (9.5.1.4).

26 An object may have the ASYNCHRONOUS attribute in a particular scoping unit without necessarily  
27 having it in other scoping units (11.2.1, 16.5.1.4). If an object has the ASYNCHRONOUS attribute,  
28 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.

### 1 **5.3.5 BIND attribute for data entities**

2 The BIND attribute for a variable or common block specifies that it is capable of interoperating with a  
3 C variable that has external linkage (15.4).

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

5 C517 An entity with the BIND attribute shall be a common block, variable, or procedure.

6 C518 A variable with the BIND attribute shall be declared in the specification part of a module.

7 C519 A variable with the BIND attribute shall be interoperable (15.3).

8 C520 Each variable of a common block with the BIND attribute shall be interoperable.

9 C521 (R508) The *scalar-char-initialization-expr* shall be of default character kind. If the value of the  
10 *scalar-char-initialization-expr* after discarding leading and trailing blanks has nonzero length,  
11 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.

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

### 14 **5.3.6 CONTIGUOUS attribute**

15 C522 An entity that has the CONTIGUOUS attribute shall be an array pointer or an assumed-shape  
16 array.

17 The **CONTIGUOUS attribute** specifies that an assumed-shape array can only be argument associated  
18 with a contiguous object, or that an array pointer can only be pointer associated with a contiguous target.

19 An object is **contiguous** if it is not the real or imaginary part of an array of type complex, and is:

- 20 (1) an object with the CONTIGUOUS attribute,
- 21 (2) a scalar object,
- 22 (3) a nonpointer array that is not assumed-shape,
- 23 (4) an array allocated by an ALLOCATE statement,
- 24 (5) an assumed-shape array that is argument associated with an array that is contiguous,
- 25 (6) a pointer associated with a contiguous target,

- 1 (7) an array with at most one element, or
- 2 (8) a nonzero-sized array section (6.2.2) with the following properties:
- 3 (a) Its base object is contiguous.
- 4 (b) It does not have a vector subscript.
- 5 (c) The elements of the section, in array element order, are a subset of the base object
- 6 elements that are consecutive in array element order.
- 7 (d) If the array is of type character and a *substring-range* appears, the *substring-range*
- 8 specifies all of the characters of the *parent-string* (6.1.1).
- 9 (e) Only its final *part-ref* has nonzero rank.

### J3 internal note

#### Unresolved Technical Issue 11

The above list (CONTIGUOUS definition) does not conform to the ISO guidelines.

- 10 An object is not contiguous if it is an array subobject, and
- 11 (1) the object has two or more elements,
- 12 (2) the elements of the object in array element order are not consecutive in the elements of the
- 13 base object,
- 14 (3) the object is not of type character with length zero, and
- 15 (4) the object is not of a derived type that has no ultimate components other than zero-sized
- 16 arrays and characters with length zero.
- 17 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 allows 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

## 18 5.3.7 DIMENSION attribute

### 19 5.3.7.1 General

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

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

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

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

24 The **co-bounds** are given within square brackets in the co-array's declaration or allocation. If the co-

25 rank is  $r$ , the co-array has lower co-bounds  $l_i$ ,  $i = 1, 2, \dots, r$ , upper co-bounds  $u_i$ ,  $i = 1, 2, \dots, r - 1$ , and

26 co-extents  $u_i - l_i + 1$ ,  $i = 1, 2, \dots, r - 1$ .

**J3 internal note****Unresolved Technical Issue 12**

The last sentence "... the co-array has lower co-bounds ..." is meaningless gibberish. Whence came these  $l_i$  et al? Not from around here as far as I see...

Furthermore, it probably belongs somewhere else. Why is there not a specific subclause for co-array declarations? Surely if assumed-size arrays warrant their own subclause, co-arrays ought not to be lumped in with the general part of arrays.

**NOTE 5.9**

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

See Note 12.28 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 C523 (R501) A co-array that has the ALLOCATABLE attribute shall be specified with a *co-array-spec*  
 4        that is a *deferred-shape-spec-list*.
- 5 C524 A co-array shall be a component or a variable that is not a function result.
- 6 C525 An entity whose type has a co-array ultimate component shall be a nonpointer nonallocatable  
 7        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.

An allocatable component of a co-array need be allocated on an image only if it is referenced or defined.

- 8 R510 *array-spec*                    **is** *explicit-shape-spec-list*  
 9                                            **or** *assumed-shape-spec-list*  
 10                                           **or** *deferred-shape-spec-list*  
 11                                           **or** *assumed-size-spec*  
 12                                           **or** *implied-shape-spec-list*
- 13 R511 *co-array-spec*                **is** *deferred-shape-spec-list*  
 14                                           **or** *assumed-size-spec*

**J3 internal note****Unresolved Technical Issue 13**

This does not work, because there are no semantics for the use of the syntax terms *deferred-shape-spec-list* and *assumed-size-spec* out of their *array-spec* context. (In so far as there are semantics, they specify array bounds not co-array co-bounds.)

- 15 C526 The sum of the rank and co-rank of an entity shall not exceed fifteen.

**NOTE 5.11**

Examples of DIMENSION attribute specifications are:

```
SUBROUTINE EX (N, A, B)
  REAL, DIMENSION (N, 10) :: W           ! Automatic explicit-shape array
  REAL A (:), B (0:)                     ! Assumed-shape arrays
```

## NOTE 5.11 (cont.)

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	<i>explicit-shape-spec</i>	<b>is</b>	[ <i>lower-bound</i> : ] <i>upper-bound</i>
5 R513	<i>lower-bound</i>	<b>is</b>	<i>specification-expr</i>
6 R514	<i>upper-bound</i>	<b>is</b>	<i>specification-expr</i>

7 C527 (R512) An *explicit-shape-spec* whose bounds are not initialization expressions shall appear only  
 8 in a subprogram or interface body.

9 If an explicit-shape array has bounds that are not initialization expressions, the bounds, and hence  
 10 shape, are determined at entry to the procedure by evaluating the bounds expressions. The bounds of  
 11 such an array are unaffected by the redefinition or undefinition of any variable during execution of the  
 12 procedure.

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

20 **5.3.7.3 Assumed-shape array**

21 An **assumed-shape array** is a nonpointer dummy argument array that takes its shape from the asso-  
 22 ciated actual argument array.

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

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

25 The extent of a dimension of an assumed-shape array dummy argument is the extent of the corresponding  
 26 dimension of the associated actual argument array. If the lower bound value is  $d$  and the extent of the  
 27 corresponding dimension of the associated actual argument array is  $s$ , then the value of the upper bound  
 28 is  $s + d - 1$ . If *lower-bound* appears it specifies the lower bound; otherwise the lower bound is 1.

29 **5.3.7.4 Deferred-shape array**

30 A **deferred-shape array** is an allocatable array or an array pointer.

31 An **allocatable array** is an array that has the ALLOCATABLE attribute and a specified rank, but its  
 32 bounds, and hence shape, are determined by allocation or argument association.

33 An **array pointer** is an array with the POINTER attribute and a specified rank. Its bounds, and hence  
 34 shape, are determined when it is associated with a target.

1 R516 *deferred-shape-spec* is :

2 C528 An array that has the POINTER or ALLOCATABLE attribute shall have an *array-spec* that is  
3 a *deferred-shape-spec-list*.

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

5 The size, bounds, and shape of an unallocated allocatable array or a disassociated array pointer are  
6 undefined. No part of such an array shall be referenced or defined; however, the array may appear as an  
7 argument to an intrinsic inquiry function as specified in 13.1.

8 The bounds of each dimension of an allocatable array are those specified when the array is allocated.

9 The bounds of each dimension of an array pointer may be specified in two ways:

- 10 (1) in an ALLOCATE statement (6.3.1) when the target is allocated;
- 11 (2) by pointer assignment (7.4.2).

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

#### 14 5.3.7.5 Assumed-size array

15 An **assumed-size array** is a dummy argument array whose size is assumed from that of an associated  
16 actual argument. The rank and extents may differ for the actual and dummy arrays; only the size of the  
17 actual array is assumed by the dummy array. An assumed-size array is declared with an *assumed-size-*  
18 *spec*.

19 R517 *assumed-size-spec* is [ *explicit-shape-spec-list* , ] [ *lower-bound* : ] \*

20 C529 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy  
21 data argument.

22 C530 An assumed-size array with INTENT (OUT) shall not be polymorphic, of a finalizable type, of  
23 a type with an ultimate allocatable component, or of a type for which default initialization is  
24 specified.

25 The size of an assumed-size array is determined as follows.

- 26 (1) If the actual argument associated with the assumed-size dummy array is an array of any  
27 type other than default character, the size is that of the actual array.
- 28 (2) If the actual argument associated with the assumed-size dummy array is an array element  
29 of any type other than default character with a subscript order value of  $r$  (6.2.2.2) in an  
30 array of size  $x$ , the size of the dummy array is  $x - r + 1$ .
- 31 (3) If the actual argument is a default character array, default character array element, or a  
32 default character array element substring (6.1.1), and if it begins at character storage unit  $t$   
33 of an array with  $c$  character storage units, the size of the dummy array is  $\text{MAX}(\text{INT}((c -$   
34  $t + 1)/e), 0)$ , where  $e$  is the length of an element in the dummy character array.
- 35 (4) If the actual argument is of type default character and is a scalar that is not an array element  
36 or array element substring designator, the size of the dummy array is  $\text{MAX}(\text{INT}(l/e), 0)$ ,  
37 where  $e$  is the length of an element in the dummy character array and  $l$  is the length of the  
38 actual argument.

39 The rank is equal to one plus the number of *explicit-shape-specs*.

40 An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last  
41 dimension and no shape. An assumed-size array name shall not be written as a whole array reference  
42 except as an actual argument in a procedure reference for which the shape is not required.

1 If an *explicit-shape-spec-list* appears, it specifies the bounds of the first rank  $-1$  dimensions. If *lower-*  
 2 *bound* appears it specifies the lower bound of the last dimension; otherwise that lower bound is 1. An  
 3 assumed-size array may be subscripted or sectioned (6.2.2.3). The upper bound shall not be omitted  
 4 from a subscript triplet in the last dimension.

5 If an assumed-size array has bounds that are not initialization expressions, the bounds are determined  
 6 at entry to the procedure. The bounds of such an array are unaffected by the redefinition or undefinition  
 7 of any variable during execution of the procedure.

#### 8 **5.3.7.6 Implied-shape array**

9 An **implied-shape array** is a named constant that takes its shape from the *initialization-expr* in its  
 10 declaration. An implied-shape array is declared with an *implied-shape-spec-list*.

11 R518 *implied-shape-spec*                      **is** [ *lower-bound* : ] \*

12 C531 An implied-shape array shall be a named constant.

13 The rank of an implied-shape array is the number of *implied-shape-specs* in the *implied-shape-spec-list*.

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

#### 17 **5.3.8 EXTERNAL attribute**

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

20 C532 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.

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

23 If an external procedure or dummy procedure is used as an actual argument or is the target of a procedure  
 24 pointer assignment, it shall be declared to have the EXTERNAL attribute.

25 A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

#### 26 **5.3.9 INTENT attribute**

27 The **INTENT attribute** specifies the intended use of a dummy argument. An INTENT (IN) dummy  
 28 argument is suitable for receiving data from the invoking scoping unit, an INTENT (OUT) dummy  
 29 argument is suitable for returning data to the invoking scoping unit, and an INTENT (INOUT) dummy  
 30 argument is suitable for use both to receive data from and to return data to the invoking scoping unit.

31 R519 *intent-spec*                              **is** IN  
 32                                                      **or** OUT  
 33                                                      **or** INOUT

34 C533 An entity with the INTENT attribute shall be a dummy data object or a dummy procedure  
 35 pointer.

36 C534 (R519) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable  
 37 definition context (16.6.7).

38 C535 A pointer with the INTENT (IN) attribute shall not appear in a pointer association context

1 (16.6.8).

2 The INTENT (IN) attribute for a nonpointer dummy argument specifies that it shall neither be de-  
3 fined nor become undefined during the execution of the procedure. The INTENT (IN) attribute for a  
4 pointer dummy argument specifies that during the execution of the procedure its association shall not  
5 be changed except that it may become undefined if the target is deallocated other than through the  
6 pointer (16.5.2.2.3).

7 The INTENT (OUT) attribute for a nonpointer dummy argument specifies that the dummy argument  
8 becomes undefined on invocation of the procedure, except for any subcomponents that are default-  
9 initialized (4.5.4.5). Any actual argument that becomes associated with such a dummy argument shall  
10 be definable. The INTENT (OUT) attribute for a pointer dummy argument specifies that on invocation  
11 of the procedure the pointer association status of the dummy argument becomes undefined. Any actual  
12 argument associated with such a pointer dummy shall be a pointer variable.

#### NOTE 5.12

If the actual argument is finalizable it will be finalized before undefinition and default initialization of the dummy argument (4.5.6).

13 The INTENT (INOUT) attribute for a nonpointer dummy argument specifies that any actual argument  
14 associated with the dummy argument shall be definable. The INTENT (INOUT) attribute for a pointer  
15 dummy argument specifies that any actual argument associated with the dummy argument shall be a  
16 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).

17 If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the  
18 argument associated entity (12.5.1.4, 12.5.1.6, 12.5.1.7).

#### 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
```

19 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
```

**NOTE 5.15 (cont.)**

is prohibited, but

$X\%P = 0$

is allowed (provided that  $X\%P$  is associated with a definable target).

Similarly, the **INTENT** restrictions on pointer dummy arguments apply only to the association of the dummy argument; they do not restrict the operations allowed on its target.

**NOTE 5.16**

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 there is any possibility that 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. Because an **INTENT(OUT)** variable is considered undefined on entry to the procedure, any default initialization specified for its type will be applied.

**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 C536 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 (12.4.2.1) 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 16.3.4.

### 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 (12.5.1.9). The **PRESENT** intrinsic function can be used to



- 1 determine whether an optional dummy argument is associated with an actual argument.
- 2 C537 An entity with the OPTIONAL attribute shall be a dummy argument.

### 3 5.3.12 PARAMETER attribute

- 4 The **PARAMETER attribute** specifies that an entity is a named constant. The entity has the value
- 5 specified by its *initialization-expr*, converted, if necessary, to the type, type parameters and shape of the
- 6 entity.
- 7 C538 An entity with the PARAMETER attribute shall not be a variable, a co-array, or a procedure.
- 8 A named constant shall not be referenced unless it has been defined previously in the same statement,
- 9 defined in a prior statement, or made accessible by use or host association.

#### NOTE 5.17

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 ( ))
```

### 10 5.3.13 POINTER attribute

- 11 Entities with the **POINTER attribute** can be associated with different data objects or procedures
- 12 during execution of a program. A pointer is either a data pointer or a procedure pointer. Procedure
- 13 pointers are described in 12.4.2.3.
- 14 C539 An entity with the POINTER attribute shall not have the ALLOCATABLE, INTRINSIC, or
- 15 TARGET attribute.
- 16 C540 A procedure with the POINTER attribute shall have the EXTERNAL attribute.
- 17 C541 A co-array shall not have the POINTER attribute.
- 18 A data pointer shall not be referenced unless it is pointer associated with a target object that is defined.
- 19 A data pointer shall not be defined unless it is pointer associated with a target object that is definable.
- 20 If a data pointer is associated, the values of its deferred type parameters are the same as the values of
- 21 the corresponding type parameters of its target.
- 22 A procedure pointer shall not be referenced unless it is pointer associated with a target procedure.

#### NOTE 5.18

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.

### 1 5.3.14 PROTECTED attribute

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

3 C542 The PROTECTED attribute shall be specified only in the specification part of a module.

4 C543 An entity with the PROTECTED attribute shall be a procedure pointer or variable.

5 C544 An entity with the PROTECTED attribute shall not be in a common block.

6 C545 A nonpointer object that has the PROTECTED attribute and is accessed by use association  
7 shall not appear in a variable definition context (16.6.7) or as the *data-target* or *proc-target* in  
8 a *pointer-assignment-stmt*.

9 C546 A pointer that has the PROTECTED attribute and is accessed by use association shall not  
10 appear in a pointer association context (16.6.8).

11 Other than within the module in which an entity is given the PROTECTED attribute, or within any of  
12 its descendants,

13 (1) if it is a nonpointer object, it is not definable, and

14 (2) if it is a pointer, its association status shall not be changed except that it may become  
15 undefined if its target is deallocated other than through the pointer (16.5.2.2.3) or if its  
16 target becomes undefined by execution of a RETURN or END statement.

17 If an object has the PROTECTED attribute, all of its subobjects have the PROTECTED attribute.

#### NOTE 5.19

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.

### 18 5.3.15 SAVE attribute

19 The SAVE attribute specifies that a variable retains its association status, allocation status, definition  
20 status, and value after execution of a RETURN or END statement unless it is a pointer and its target  
21 becomes undefined (16.5.2.2.3(4)). If it is a local variable of a subprogram it is shared by all instances  
22 (12.6.2.3) of the subprogram.

23 Giving a common block the SAVE attribute confers the attribute on all variables in the common block.

24 C547 An entity with the SAVE attribute shall be a common block, variable, or procedure pointer.

25 C548 The SAVE attribute shall not be specified for a dummy argument, a function result, an automatic

- 1 data object, or an object that is in a common block.
- 2 C549 The SAVE attribute shall be specified for a co-array unless it is a dummy argument, declared
- 3 in the main program, or allocatable and declared in a non-recursive procedure.

**NOTE 5.20**

An allocatable co-array is required to have the SAVE attribute in a recursive procedure because of the difficulties associated with defining the separate levels of recursion. Without the SAVE attribute, each would have a separate instance of the co-array and implicit synchronization would be required for each allocation and deallocation of each instance.

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 not dummy arguments are required to have the SAVE attribute because otherwise they might be implemented as if they were automatic co-arrays.

**J3 internal note****Unresolved Technical Issue 8**

This text about automatic co-arrays not being permitted because of putative implicit synchronisation absolutely does **NOT** belong here (in the SAVE attribute), nor did it belong where 06-174r3 said to insert it (in the middle of a 2-page list of constraints!). Co-arrays probably need their own subclause and it can go there. Pity they didn't use the CODIMENSION keyword instead of hijacking the DIMENSION keyword, then it would be a lot tidier with things having a rather obvious home.

Furthermore, the last sentence is incorrect, because not all explicit-shape co-arrays are required to have the SAVE attribute, viz ones declared in the main program.

**J3 internal note****Unresolved Technical Issue 9**

The note says that allocatable co-arrays have to be (inconsistently) saved, "because of the difficulties associated with defining the separate levels of recursion". That is utter nonsense. 12.5.2.3 "Instances of a subprogram" was in F2003 already. If there were such difficulties, we wouldn't be able to have host association in recursive procedures (there since F90), or to pass internal procedures (of recursive procedures) as actual arguments (new in F2008).

- 4 C550 (R501) If the type specified has a co-array ultimate component, each entity in the *entity-decl-list*
- 5 shall have the SAVE attribute unless it is a dummy argument, declared in the main program,
- 6 or declared in a non-recursive procedure.

**J3 internal note**

Unresolved Technical Issue 10  
This is inconsistent.

- (1) We require the user to **SAVE** variables in non-recursive procedures if he wants the value to persist.
- (2) **ALLOCATABLE** arrays that are unsaved get deallocated in non-recursive procedures, not saved.
- (3) It requires **SAVE** in a **MODULE**, but not in a non-recursive procedure!?!  
If you are going to special-case **ANYTHING**, doing it for modules must be top priority, but it's better to avoid it entirely.

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. If a common block has the  
4 **SAVE** attribute in any scoping unit that is not a main program, it shall have the **SAVE** attribute in  
5 every scoping unit that is not a main program.

### 6 **5.3.16 TARGET attribute**

7 The **TARGET attribute** specifies that a data object may have a pointer associated with it (7.4.2).  
8 An object without the **TARGET** attribute shall not have an accessible pointer associated with it.

9 C551 An entity with the **TARGET** attribute shall be a variable.

10 C552 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.

### 1 5.3.17 VALUE attribute

2 The **VALUE attribute** specifies a type of argument association (12.5.1.4) for a dummy argument.

3 C553 An entity with the VALUE attribute shall be a scalar dummy data object.

4 C554 An entity with the VALUE attribute shall not have the ALLOCATABLE, INTENT(INOUT),  
5 INTENT(OUT), POINTER, or VOLATILE attributes.

6 C555 If an entity has the VALUE attribute, any length type parameter value in its declaration shall  
7 be omitted or specified by an initialization expression.

### 8 5.3.18 VOLATILE attribute

9 The **VOLATILE attribute** specifies that an object may be referenced, defined, or become undefined,  
10 by means not specified by the program, or by another image without synchronization.

11 C556 An entity with the VOLATILE attribute shall be a variable that is not an INTENT(IN) dummy  
12 argument.

13 An object may have the VOLATILE attribute in a particular scoping unit without necessarily having  
14 it in other scoping units (11.2.1, 16.5.1.4). If an object has the VOLATILE attribute, then all of its  
15 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.

16 A pointer with the VOLATILE attribute may additionally have its association status, dynamic type and  
17 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.

18 An allocatable object with the VOLATILE attribute may additionally have its allocation status, dynamic  
19 type and type parameters, and array bounds changed by means not specified by the program.

## 20 5.4 Attribute specification statements

### 21 5.4.1 Accessibility statements

22 R520 *access-stmt* **is** *access-spec* [ [ :: ] *access-id-list* ]

23 R521 *access-id* **is** *use-name*

24 **or** *generic-spec*

25 C557 (R520) An *access-stmt* shall appear only in the *specification-part* of a module. Only one ac-  
26 cessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a

1 module.

2 C558 (R521) Each *use-name* shall be the name of a named variable, procedure, derived type, named  
3 constant, namelist group, or macro.

4 An *access-stmt* with an *access-id-list* specifies the accessibility attribute (5.3.2), PUBLIC or PRIVATE,  
5 of each *access-id* in the list. An *access-stmt* without an *access-id* list specifies the default accessibility  
6 that applies to all potentially accessible identifiers in the *specification-part* of the module. The statement

7 PUBLIC

8 specifies a default of public accessibility. The statement

9 PRIVATE

10 specifies a default of private accessibility. If no such statement appears in a module, the default is public  
11 accessibility.

#### NOTE 5.26

Examples of accessibility statements are:

```
MODULE EX
  PRIVATE
  PUBLIC :: A, B, C, ASSIGNMENT (=), OPERATOR (+)
```

### 12 5.4.2 ALLOCATABLE statement

13 R522 *allocatable-stmt* is ALLOCATABLE [ :: ] *allocatable-decl-list*

14 R523 *allocatable-decl* is *object-name* [ ( *array-spec* ) ] ■

15 ■ [ *lbracket co-array-spec rbracket* ]

16 This 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
```

### 17 5.4.3 ASYNCHRONOUS statement

18 R524 *asynchronous-stmt* is ASYNCHRONOUS [ :: ] *object-name-list*

19 The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute (5.3.4) for a list of objects.

### 20 5.4.4 BIND statement

21 R525 *bind-stmt* is *language-binding-spec* [ :: ] *bind-entity-list*

22 R526 *bind-entity* is *entity-name*

23 or / *common-block-name* /

24 C559 (R525) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of  
25 a single *bind-entity*.

26 The BIND statement specifies the BIND attribute (5.3.5) for a list of variables and common blocks.

#### 1 5.4.5 CONTIGUOUS statement

2 R527 *contiguous-stmt* is CONTIGUOUS [ :: ] *object-name-list*

3 The CONTIGUOUS statement specifies the CONTIGUOUS attribute (5.3.6) for a list of objects.

#### 4 5.4.6 DATA statement

5 R528 *data-stmt* is DATA *data-stmt-set* [ [ , ] *data-stmt-set* ] ...

6 This statement specifies explicit initialization (5.2.2).

7 A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If a  
8 nonpointer object has been specified with default initialization in a type definition, it shall not appear  
9 in a *data-stmt-object-list*.

10 A variable that appears in a DATA statement and has not been typed previously may appear in a  
11 subsequent type declaration only if that declaration confirms the implicit typing. An array name,  
12 array section, or array element that appears in a DATA statement shall have had its array properties  
13 established by a previous specification statement.

14 Except for variables in named common blocks, a named variable has the SAVE attribute if any part of  
15 it is initialized in a DATA statement, and this may be confirmed by explicit specification.

16 R529 *data-stmt-set* is *data-stmt-object-list* / *data-stmt-value-list* /  
17 R530 *data-stmt-object* is *variable*  
18 or *data-implied-do*  
19 R531 *data-implied-do* is ( *data-i-do-object-list* , *data-i-do-variable* = ■  
20 ■ *scalar-int-initialization-expr* , ■  
21 ■ *scalar-int-initialization-expr* ■  
22 ■ [ , *scalar-int-initialization-expr* ] )  
23 R532 *data-i-do-object* is *array-element*  
24 or *scalar-structure-component*  
25 or *data-implied-do*  
26 R533 *data-i-do-variable* is *do-variable*

27 C560 A *data-stmt-object* or *data-i-do-object* shall not be a co-indexed variable.

28 C561 (R530) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring start-  
29 ing point, and substring ending point shall be an initialization expression.

30 C562 (R530) A variable whose designator appears as a *data-stmt-object* or a *data-i-do-object* shall  
31 not be a dummy argument, made accessible by use association or host association, in a named  
32 common block unless the DATA statement is in a block data program unit, in a blank common  
33 block, a function name, a function result name, an automatic object, or an allocatable variable.

34 C563 (R530) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object  
35 designator in which a pointer appears other than as the entire rightmost *part-ref*.

36 C564 (R532) The *array-element* shall be a variable.

37 C565 (R532) The *scalar-structure-component* shall be a variable.

38 C566 (R532) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *sub-*  
39 *script-list*.

40 C567 (R532) In an *array-element* or *scalar-structure-component* that is a *data-i-do-object*, any sub-  
41 script shall be an initialization expression, and any primary within that subscript that is a

1        *data-i-do-variable* shall be a DO variable of this *data-implied-do* or of a containing *data-implied-*  
2        *do*.

3    R534    *data-stmt-value*                    **is**   [ *data-stmt-repeat* \* ] *data-stmt-constant*

4    R535    *data-stmt-repeat*                **is**   *scalar-int-constant*

5                                                **or**   *scalar-int-constant-subobject*

6    C568    (R535) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named con-  
7                                                stant, it shall have been declared previously in the scoping unit or made accessible by use  
8                                                association or host association.

9    R536    *data-stmt-constant*                **is**   *scalar-constant*

10                                                **or**   *scalar-constant-subobject*

11                                                **or**   *signed-int-literal-constant*

12                                                **or**   *signed-real-literal-constant*

13                                                **or**   *null-init*

14                                                **or**   *initial-data-target*

15                                                **or**   *structure-constructor*

16    C569    (R536) If a DATA statement constant value is a named constant or a structure constructor, the  
17                                                named constant or derived type shall have been declared previously in the scoping unit or made  
18                                                accessible by use or host association.

19    C570    (R536) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.

20    R537    *int-constant-subobject*            **is**   *constant-subobject*

21    C571    (R537) *int-constant-subobject* shall be of type integer.

22    R538    *constant-subobject*                **is**   *designator*

23    C572    (R538) *constant-subobject* shall be a subobject of a constant.

24    C573    (R538) Any subscript, substring starting point, or substring ending point shall be an initializa-  
25                                                tion expression.

26    The *data-stmt-object-list* is expanded to form a sequence of pointers and scalar variables, referred to  
27    as “sequence of variables” in subsequent text. A nonpointer array whose unqualified name appears  
28    as a *data-stmt-object* or *data-i-do-object* is equivalent to a complete sequence of its array elements in  
29    array element order (6.2.2.2). An array section is equivalent to the sequence of its array elements in  
30    array element order. A *data-implied-do* is expanded to form a sequence of array elements and structure  
31    components, under the control of the *data-i-do-variable*, as in the DO construct (8.1.7.5).

32    The *data-stmt-value-list* is expanded to form a sequence of *data-stmt-constants*. A *data-stmt-repeat*  
33    indicates the number of times the following *data-stmt-constant* is to be included in the sequence; omission  
34    of a *data-stmt-repeat* has the effect of a repeat factor of 1.

35    A zero-sized array or a *data-implied-do* with an iteration count of zero contributes no variables to the  
36    expanded sequence of variables, but a zero-length scalar character variable does contribute a variable  
37    to the expanded sequence. A *data-stmt-constant* with a repeat factor of zero contributes no *data-stmt-*  
38    *constants* to the expanded sequence of scalar *data-stmt-constants*.

39    The expanded sequences of variables and *data-stmt-constants* are in one-to-one correspondence. Each  
40    *data-stmt-constant* specifies the initial value, initial data target, or *null-init* for the corresponding vari-  
41    able. The lengths of the two expanded sequences shall be the same.

42    A *data-stmt-constant* shall be *null-init* or *initial-data-target* if and only if the corresponding *data-stmt-*  
43    *object* has the POINTER attribute. If *data-stmt-constant* is *null-init*, the initial association status of



- 1 the corresponding data statement object is disassociated. If *data-stmt-constant* is *initial-data-target* the  
 2 corresponding data statement object shall be data-pointer-initialization compatible with the initial data  
 3 target; the data statement object is initially associated with the target.
- 4 A *data-stmt-constant* other than *null-init* or *initial-data-target* shall be compatible with its corresponding  
 5 variable according to the rules of intrinsic assignment (7.4.1.2). The variable is initially defined with  
 6 the value specified by the *data-stmt-constant*; if necessary, the value is converted according to the rules  
 7 of intrinsic assignment (7.4.1.3) to a value that agrees in type, type parameters, and shape with the  
 8 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.

**9 5.4.7 DIMENSION statement**

- 10 R539 *dimension-stmt*                    **is** DIMENSION [ :: ] *dimension-decl-list*  
 11 R540 *dimension-decl*                    **is** *array-name* ( *array-spec* )  
 12                                            **or** *co-array-name* [ ( *array-spec* ) ] *lbracket co-array-spec rbracket*  
 13

- 14 This 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 (:)
```

**15 5.4.8 INTENT statement**

- 16 R541 *intent-stmt*                                    **is** INTENT ( *intent-spec* ) [ :: ] *dummy-arg-name-list*  
 17 This 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
```

#### 1 **5.4.9 OPTIONAL statement**

2 R542 *optional-stmt*                      **is** OPTIONAL [ :: ] *dummy-arg-name-list*

3 This statement specifies the OPTIONAL attribute (5.3.11) for the dummy arguments 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 R543 *parameter-stmt*                      **is** PARAMETER ( *named-constant-def-list* )

8 R544 *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 (7.4.1.3) 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 R545 *pointer-stmt*                      **is** POINTER [ :: ] *pointer-decl-list*

18 R546 *pointer-decl*                      **is** *object-name* [ ( *deferred-shape-spec-list* ) ]

19                                              **or** *proc-entity-name*

20 This 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 (:, :)
```

#### 1 5.4.12 PROTECTED statement

2 R547 *protected-stmt* is PROTECTED [ :: ] *entity-name-list*

3 The **PROTECTED statement** specifies the PROTECTED attribute (5.3.14) for a list of entities.

#### 4 5.4.13 SAVE statement

5 R548 *save-stmt* is SAVE [ [ :: ] *saved-entity-list* ]

6 R549 *saved-entity* is *object-name*

7 or *proc-pointer-name*

8 or / *common-block-name* /

9 R550 *proc-pointer-name* is *name*

10 C574 (R548) If a SAVE statement with an omitted saved entity list appears in a scoping unit, no  
11 other appearance of the SAVE *attr-spec* or SAVE statement is permitted in that scoping unit.

12 A SAVE statement with a saved entity list specifies the SAVE attribute (5.3.15) for a list of entities. A  
13 SAVE statement without a saved entity list is treated as though it contained the names of all allowed  
14 items in the same scoping unit.

##### NOTE 5.34

An example of a SAVE statement is:

```
SAVE A, B, C, / BLOCKA /, D
```

#### 15 5.4.14 TARGET statement

16 R551 *target-stmt* is TARGET [ :: ] *target-decl-list*

17 R552 *target-decl* is *object-name* [ ( *array-spec* ) ] ■

18 ■ [ *lbracket co-array-spec rbracket* ]

19 This 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
```

#### 20 5.4.15 VALUE statement

21 R553 *value-stmt* is VALUE [ :: ] *dummy-arg-name-list*

22 The VALUE statement specifies the VALUE attribute (5.3.17) for a list of dummy arguments.

#### 23 5.4.16 VOLATILE statement

24 R554 *volatile-stmt* is VOLATILE [ :: ] *object-name-list*

25 The VOLATILE statement specifies the VOLATILE attribute (5.3.18) for a list of objects.

## 1 5.5 IMPLICIT statement

- 2 In a scoping unit, an **IMPLICIT statement** specifies a type, and possibly type parameters, for all  
 3 implicitly typed data entities whose names begin with one of the letters specified in the statement.  
 4 Alternatively, it may indicate that no implicit typing rules are to apply in a particular scoping unit.

5 R555 *implicit-stmt*                    **is** IMPLICIT *implicit-spec-list*  
 6                                            **or** IMPLICIT NONE  
 7 R556 *implicit-spec*                   **is** *declaration-type-spec* ( *letter-spec-list* )  
 8 R557 *letter-spec*                    **is** *letter* [ – *letter* ]

- 9 C575 (R555) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER  
 10 statements that appear in the scoping unit and there shall be no other IMPLICIT statements  
 11 in the scoping unit.

- 12 C576 (R557) If the minus and second *letter* appear, the second letter shall follow the first letter  
 13 alphabetically.

- 14 A *letter-spec* consisting of two *letters* separated by a minus is equivalent to writing a list containing all  
 15 of the letters in alphabetical order in the alphabetic sequence from the first letter through the second  
 16 letter. For example, A–C is equivalent to A, B, C. The same letter shall not appear as a single letter, or  
 17 be included in a range of letters, more than once in all of the IMPLICIT statements in a scoping unit.

- 18 In each scoping unit, there is a mapping, which may be null, between each of the letters A, B, ..., Z  
 19 and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in  
 20 its *letter-spec-list*. IMPLICIT NONE specifies the null mapping for all the letters. If a mapping is not  
 21 specified for a letter, the default for a program unit or an interface body is default integer if the letter  
 22 is I, J, ..., or N and default real otherwise, and the default for an internal or module procedure is the  
 23 mapping in the host scoping unit.

- 24 Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic  
 25 function, and is not made accessible by use association or host association is declared implicitly to be of  
 26 the type (and type parameters) mapped from the first letter of its name, provided the mapping is not  
 27 null. The mapping for the first letter of the data entity shall either have been established by a prior  
 28 IMPLICIT statement or be the default mapping for the letter. The mapping may be to a derived type  
 29 that is inaccessible in the local scope if the derived type is accessible in the host scoping unit. The data  
 30 entity is treated as if it were declared in an explicit type declaration in the outermost scoping unit in  
 31 which it appears. An explicit type specification in a FUNCTION statement overrides an IMPLICIT  
 32 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.
    ...
```

**NOTE 5.36 (cont.)**

```

    END FUNCTION JFUN
END MODULE EXAMPLE_MODULE
SUBROUTINE SUB
    IMPLICIT COMPLEX (C)
    C = (3.0, 2.0)      ! C is implicitly declared COMPLEX
    ...
CONTAINS
    SUBROUTINE SUB1
        IMPLICIT INTEGER (A, C)
        C = (0.0, 0.0)  ! C is host associated and of
                        ! type complex
        Z = 1.0         ! Z is implicitly declared REAL
        A = 2           ! A is implicitly declared INTEGER
        CC = 1          ! CC is implicitly declared INTEGER
        ...
    END SUBROUTINE SUB1
    SUBROUTINE SUB2
        Z = 2.0         ! Z is implicitly declared REAL and
                        ! is different from the variable of
                        ! the same name in SUB1
        ...
    END SUBROUTINE SUB2
    SUBROUTINE SUB3
        USE EXAMPLE_MODULE ! Accesses integer function FUN
                        ! by use association
        Q = FUN (K)        ! Q is implicitly declared REAL and
        ...               ! K is implicitly declared INTEGER
    END SUBROUTINE SUB3
END SUBROUTINE SUB

```

**NOTE 5.37**

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

**NOTE 5.37 (cont.)**

to type BLOB, so AA is of type BLOB.

## 1 **5.6 NAMELIST statement**

2 A **NAMELIST statement** specifies a group of named data objects, which may be referred to by a  
3 single name for the purpose of data transfer (9.5, 10.11).

```
4 R558  namelist-stmt           is  NAMELIST ■
5                                     ■ / namelist-group-name / namelist-group-object-list ■
6                                     ■ [ [ , ] / namelist-group-name / ■
7                                     ■ namelist-group-object-list ] ...
```

8 C577 (R558) The *namelist-group-name* shall not be a name accessed by use association.

```
9 R559  namelist-group-object    is  variable-name
```

10 C578 (R559) A *namelist-group-object* shall not be an assumed-size array.

11 C579 (R558) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-*  
12 *name* has the PUBLIC attribute.

13 The order in which the variables are specified in the NAMELIST statement determines the order in  
14 which the values appear on output.

15 Any *namelist-group-name* may occur more than once in the NAMELIST statements in a scoping unit.  
16 The *namelist-group-object-list* following each successive appearance of the same *namelist-group-name* in  
17 a scoping unit is treated as a continuation of the list for that *namelist-group-name*.

18 A namelist group object may be a member of more than one namelist group.

19 A namelist group object shall either be accessed by use or host association or shall have its type, type  
20 parameters, and shape specified by previous specification statements or the procedure heading in the  
21 same scoping unit or by the implicit typing rules in effect for the scoping unit. If a namelist group object  
22 is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall  
23 confirm the implied type and type parameters.

**NOTE 5.38**

An example of a NAMELIST statement is:

```
NAMELIST /NLIST/ A, B, C
```

## 24 **5.7 Storage association of data objects**

### 25 **5.7.1 EQUIVALENCE statement**

#### 26 **5.7.1.1 General**

27 An **EQUIVALENCE statement** is used to specify the sharing of storage units by two or more objects  
28 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.

- 1 If the equivalenced objects have differing type or type parameters, the EQUIVALENCE statement does  
 2 not cause type conversion or imply mathematical equivalence. If a scalar and an array are equivalenced,  
 3 the scalar does not have array properties and the array does not have the properties of a scalar.

4 R560 *equivalence-stmt*                    **is** EQUIVALENCE *equivalence-set-list*  
 5 R561 *equivalence-set*                    **is** ( *equivalence-object* , *equivalence-object-list* )  
 6 R562 *equivalence-object*                **is** *variable-name*  
 7                                                **or** *array-element*  
 8                                                **or** *substring*

- 9 C580 (R562) An *equivalence-object* shall not be a designator with a base object that is a dummy  
 10 argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ulti-  
 11 mate component, an object of a nonsequence derived type, an object of a derived type that has  
 12 a pointer at any level of component selection, an automatic object, a function name, an entry  
 13 name, a result name, a variable with the BIND attribute, a variable in a common block that  
 14 has the BIND attribute, or a named constant.
- 15 C581 (R562) An *equivalence-object* shall not be a designator that has more than one *part-ref*.
- 16 C582 (R562) An *equivalence-object* shall not be a co-array or a subobject thereof.
- 17 C583 (R562) An *equivalence-object* shall not have the TARGET attribute.
- 18 C584 (R562) Each subscript or substring range expression in an *equivalence-object* shall be an integer  
 19 initialization expression (7.1.7).
- 20 C585 (R561) If an *equivalence-object* is of type default integer, default real, double precision real,  
 21 default complex, default logical, or numeric sequence type, all of the objects in the equivalence  
 22 set shall be of these types.
- 23 C586 (R561) If an *equivalence-object* is of type default character or character sequence type, all of the  
 24 objects in the equivalence set shall be of these types.
- 25 C587 (R561) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or  
 26 character sequence type, all of the objects in the equivalence set shall be of the same type with  
 27 the same type parameter values.
- 28 C588 (R561) If an *equivalence-object* is of an intrinsic type other than default integer, default real,  
 29 double precision real, default complex, default logical, or default character, all of the objects in  
 30 the equivalence set shall be of the same type with the same kind type parameter value.
- 31 C589 (R562) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equiv-  
 32 alence set shall have the PROTECTED attribute.
- 33 C590 (R562) The name of an *equivalence-object* shall not be a name made accessible by use association.
- 34 C591 (R562) A *substring* shall not have length zero.

**NOTE 5.40**

The EQUIVALENCE statement allows the equivalencing of sequence structures and the equiv-  
 alencing of objects of intrinsic type with nondefault type parameters, but there are strict rules  
 regarding the appearance of these objects in an EQUIVALENCE statement.

**NOTE 5.40 (cont.)**

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, or default logical type such that components of the structure ultimately 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 the equivalenced character objects need  
 11 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)	---	



#### 1 5.7.1.4 Array names and array element designators

2 For a nonzero-sized array, the use of the array name unqualified by a subscript list as an *equivalence-*  
 3 *object* has the same effect as using an array element designator that identifies the first element of the  
 4 array.

#### 5 5.7.1.5 Restrictions on EQUIVALENCE statements

6 An EQUIVALENCE statement shall not specify that the same storage unit is to occur more than once  
 7 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).

8 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
```

#### 9 5.7.2 COMMON statement

##### 10 5.7.2.1 General

11 The **COMMON statement** specifies blocks of physical storage, called **common blocks**, that can be  
 12 accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global  
 13 data facility based on storage association (16.5.3).

14 The common blocks specified by the COMMON statement may be named and are called **named com-**  
 15 **mon blocks**, or may be unnamed and are called **blank common**.

16	R563	<i>common-stmt</i>	<b>is</b>	<b>COMMON</b> ■
17				■ [ / [ <i>common-block-name</i> ] / ] <i>common-block-object-list</i> ■
18				■ [ [ , ] / [ <i>common-block-name</i> ] / ■
19				■ <i>common-block-object-list</i> ] ...
20	R564	<i>common-block-object</i>	<b>is</b>	<i>variable-name</i> [ ( <i>array-spec</i> ) ]
21			<b>or</b>	<i>proc-pointer-name</i>

22 C592 (R564) An *array-spec* in a *common-block-object* shall be an *explicit-shape-spec-list*.

23 C593 (R564) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all  
 24 *common-block-object-lists* within a scoping unit.

25 C594 (R564) A *common-block-object* shall not be a dummy argument, an allocatable variable, a  
 26 derived-type object with an ultimate component that is allocatable, an automatic object, a

1 function name, an entry name, a variable with the BIND attribute, a co-array, or a result name.

2 C595 (R564) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.2) or a  
3 type with the BIND attribute and it shall have no default initialization.

4 C596 (R564) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use  
5 association.

6 In each COMMON statement, the data objects whose names appear in a common block object list  
7 following a common block name are declared to be in that common block. If the first common block  
8 name is omitted, all data objects whose names appear in the first common block object list are specified to  
9 be in blank common. Alternatively, the appearance of two slashes with no common block name between  
10 them declares the data objects whose names appear in the common block object list that follows to be  
11 in blank common.

12 Any common block name or an omitted common block name for blank common may occur more than  
13 once in one or more COMMON statements in a scoping unit. The common block list following each  
14 successive appearance of the same common block name in a scoping unit is treated as a continuation of  
15 the list for that common block name. Similarly, each blank common block object list in a scoping unit  
16 is treated as a continuation of blank common.

17 The form *variable-name* (*array-spec*) specifies the DIMENSION attribute for that variable.

18 If derived-type objects of numeric sequence type (4.5.2) or character sequence type (4.5.2) appear in  
19 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
```

### 20 5.7.2.2 Common block storage sequence

21 For each common block in a scoping unit, a **common block storage sequence** is formed as follows:

- 22 (1) A storage sequence is formed consisting of the sequence of storage units in the storage  
23 sequences (16.5.3.2) of all data objects in the common block object lists for the common  
24 block. The order of the storage sequences is the same as the order of the appearance of the  
25 common block object lists in the scoping unit.
- 26 (2) The storage sequence formed in (1) is extended to include all storage units of any storage  
27 sequence associated with it by equivalence association. The sequence shall be extended only  
28 by adding storage units beyond the last storage unit. Data objects associated with an entity  
29 in a common block are considered to be in that common block.

30 Only COMMON statements and EQUIVALENCE statements appearing in the scoping unit contribute  
31 to common block storage sequences formed in that scoping unit.

### 32 5.7.2.3 Size of a common block

33 The **size of a common block** is the size of its common block storage sequence, including any extensions  
34 of the sequence resulting from equivalence association.

#### 1 5.7.2.4 Common association

2 Within a program, the common block storage sequences of all nonzero-sized common blocks with the  
 3 same name have the same first storage unit, and the common block storage sequences of all zero-sized  
 4 common blocks with the same name are storage associated with one another. Within a program, the  
 5 common block storage sequences of all nonzero-sized blank common blocks have the same first storage  
 6 unit and the storage sequences of all zero-sized blank common blocks are associated with one another and  
 7 with the first storage unit of any nonzero-sized blank common blocks. This results in the association of  
 8 objects in different scoping units. Use association or host association may cause these associated objects  
 9 to be accessible in the same scoping unit.

10 A nonpointer object of default integer type, default real type, double precision real type, default complex  
 11 type, default logical type, or numeric sequence type shall be associated only with nonpointer objects of  
 12 these types.

13 A nonpointer object of type default character or character sequence type shall be associated only with  
 14 nonpointer objects of these types.

15 A nonpointer object of a derived type that is not a numeric sequence or character sequence type shall  
 16 be associated only with nonpointer objects of the same type with the same type parameter values.

17 A nonpointer object of intrinsic type other than default integer, default real, double precision real,  
 18 default complex, default logical, or default character shall be associated only with nonpointer objects of  
 19 the same type and type parameters.

20 A data pointer shall be storage associated only with data pointers of the same type and rank. Data  
 21 pointers that are storage associated shall have deferred the same type parameters; corresponding non-  
 22 deferred type parameters shall have the same value. A procedure pointer shall be storage associated  
 23 only with another procedure pointer; either both interfaces shall be explicit or both interfaces shall be  
 24 implicit. If the interfaces are explicit, the characteristics shall be the same. If the interfaces are implicit,  
 25 either both shall be subroutines or both shall be functions with the same type and type parameters.

26 An object with the TARGET attribute shall be storage associated only with another object that has  
 27 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).

#### 28 5.7.2.5 Differences between named common and blank common

29 A blank common block has the same properties as a named common block, except for the following.

- 30 (1) Execution of a RETURN or END statement may cause data objects in a named common  
 31 block to become undefined unless the common block has the SAVE attribute, but never  
 32 causes data objects in blank common to become undefined (16.6.6).
- 33 (2) Named common blocks of the same name shall be of the same size in all scoping units of a  
 34 program in which they appear, but blank common blocks may be of different sizes.
- 35 (3) A data object in a named common block may be initially defined by means of a DATA  
 36 statement or type declaration statement in a block data program unit (11.3), but objects in

1           blank common shall not be initially defined.

### 2   **5.7.3 Restrictions on common and equivalence**

3   An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to  
4   be associated.

5   Equivalence association shall not cause a derived-type object with default initialization to be associated  
6   with an object in a common block.

7   Equivalence association shall not cause a common block storage sequence to be extended by adding  
8   storage units preceding the first storage unit of the first object specified in a COMMON statement for  
9   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
```

## 1 6 Use of data objects

2 The appearance of a data object designator in a context that requires its value is termed a reference. A  
 3 reference is permitted only if the data object is defined. A reference to a pointer is permitted only if the  
 4 pointer is associated with a target object that is defined. A data object becomes defined with a value  
 5 when events described in 16.6.5 occur.

6 R601 *variable* **is** *designator*  
 7 **or** *expr*

8 C601 (R601) *designator* shall not be a constant or a subobject of a constant.

9 C602 (R601) The *expr* shall be a reference to a function that has a pointer result.

10 A variable is either the data object denoted by *designator* or the target of *expr*.

11 R602 *variable-name* **is** *name*

12 C603 (R602) A *variable-name* shall be the name of a variable.

13 R603 *designator* **is** *object-name*  
 14 **or** *array-element*  
 15 **or** *array-section*  
 16 **or** *complex-part-designator*  
 17 **or** *structure-component*  
 18 **or** *substring*

19 R604 *logical-variable* **is** *variable*

20 C604 (R604) *logical-variable* shall be of type logical.

21 R605 *default-logical-variable* **is** *variable*

22 C605 (R605) *default-logical-variable* shall be of type default logical.

23 R606 *char-variable* **is** *variable*

24 C606 (R606) *char-variable* shall be of type character.

25 R607 *default-char-variable* **is** *variable*

26 C607 (R607) *default-char-variable* shall be of type default character.

27 R608 *int-variable* **is** *variable*

28 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 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  
 7 array (6.2). Its value, if defined, is a single element from the set of values that characterize its type.

### NOTE 6.2

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

8 A scalar has rank zero.

### 9 6.1.1 Substrings

10 A **substring** is a contiguous portion of a character string (4.4.5). The following rules define the forms  
 11 of a substring:

12 R609	<i>substring</i>	<b>is</b>	<i>parent-string</i> ( <i>substring-range</i> )
13 R610	<i>parent-string</i>	<b>is</b>	<i>scalar-variable-name</i>
14		<b>or</b>	<i>array-element</i>
15		<b>or</b>	<i>scalar-structure-component</i>
16		<b>or</b>	<i>scalar-constant</i>
17 R611	<i>substring-range</i>	<b>is</b>	[ <i>scalar-int-expr</i> ] : [ <i>scalar-int-expr</i> ]

18 C609 (R610) *parent-string* shall be of type character.

19 The value of the first *scalar-int-expr* in *substring-range* is called the **starting point** and the value of  
 20 the second one is called the **ending point**. The length of a substring is the number of characters in the  
 21 substring and is  $\text{MAX}(l - f + 1, 0)$ , where  $f$  and  $l$  are the starting and ending points, respectively.

22 Let the characters in the parent string be numbered 1, 2, 3, ...,  $n$ , where  $n$  is the length of the parent  
 23 string. Then the characters in the substring are those from the parent string from the starting point and  
 24 proceeding in sequence up to and including the ending point. Both the starting point and the ending  
 25 point shall be within the range 1, 2, ...,  $n$  unless the starting point exceeds the ending point, in which  
 26 case the substring has length zero. If the starting point is not specified, the default value is 1. If the  
 27 ending point is not specified, the default value is  $n$ .

28 If the parent is a variable, the substring is also a variable.

### NOTE 6.3

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

### 1 6.1.2 Structure components

2 A **structure component** is part of an object of derived type; it may be referenced by an object  
3 designator. A structure component may be a scalar or an array.

4 R612 *data-ref* **is** *part-ref* [ % *part-ref* ] ...

5 R613 *part-ref* **is** *part-name* [ ( *section-subscript-list* ) ] [ *image-selector* ]

6 C610 (R612) Each *part-name* except the rightmost shall be of derived type.

7 C611 (R612) Each *part-name* except the leftmost shall be the name of a component of the declared  
8 type of the preceding *part-name*.

9 C612 (R612) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.

10 C613 (R612) The leftmost *part-name* shall be the name of a data object.

11 C614 (R613) If a *section-subscript-list* appears, the number of *section-subscripts* shall equal the rank  
12 of *part-name*.

13 C615 (R613) If *image-selector* appears and *part-name* is an array, *section-subscript-list* shall appear.

14 C616 (R613) A *data-ref* that is a co-indexed object shall not be of a type that has a pointer ultimate  
15 component.

#### NOTE 6.4

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

#### J3 internal note

##### Unresolved Technical Issue 20

Is “Z[P] = Z” allowed if Z has an allocatable component? This would imply remote (re)allocation, which I was told was bad when I asked why we were prohibiting “array[i]” forcing it to be “array(:)[i]”. Also, constraint (C633a) in ALLOCATE prevents explicit remote allocation. If the answer is that remote reallocation is fine in this instance, we need to make that consistent (including allowing explicit remote allocation). Or if it is bad, we need to remove it consistently. Subgroup say they want to allow this, but require allocatable components in Z[P] to have the same shape as those in Z, removing the auto realloc. The editor disagrees that this is an obviously “good” technical fix; see discussion under assignment.

16 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  
17 a section subscript list is the number of subscript triplets and vector subscripts in the list.

18 C617 (R612) There shall not be more than one *part-ref* with nonzero rank. A *part-name* to the right  
19 of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.

20 The rank of a *data-ref* is the rank of the *part-ref* with nonzero rank, if any; otherwise, the rank is zero.

21 The **base object** of a *data-ref* is the data object whose name is the leftmost part name.

22 The type and type parameters, if any, of a *data-ref* are those of the rightmost part name.

23 A *data-ref* with more than one *part-ref* is a subobject of its base object if none of the *part-names*,  
24 except for possibly the rightmost, are pointers. If the rightmost *part-name* is the only pointer, then the  
25 *data-ref* is a subobject of its base object in contexts that pertain to its pointer association status but

- 1 not in any other contexts.

#### NOTE 6.5

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.

- 2 R614 *structure-component* **is** *data-ref*
- 3 C618 (R614) There shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form
- 4 *part-name*.
- 5 A structure component shall be neither referenced nor defined before the declaration of the base object.
- 6 A structure component is a pointer only if the rightmost part name is defined to have the POINTER
- 7 attribute.

#### NOTE 6.6

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 6.7

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.

- 8 A **subcomponent** of an object of derived type is a component of that object or of a subobject of that
- 9 object.

### 6.1.3 Complex parts

- 11 A **complex part designator** is used to designate the real or imaginary part of a complex data object,
- 12 independently of the other part.
- 13 R615 *complex-part-designator* **is** *designator* % RE
- 14 **or** *designator* % IM
- 15 C619 (R615) The *designator* shall be of complex type.
- 16 If *complex-part-designator* is *designator*%RE it designates the real part of *designator*. If it is *designa-*
- 17 *tor*%IM it designates the imaginary part of *designator*. The type of a *complex-part-designator* is real,
- 18 and its kind and shape are those of the *designator*.



**NOTE 6.8**

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
```

### 1 6.1.4 Type parameter inquiry

2 A **type parameter inquiry** is used to inquire about a type parameter of a data object. It applies to  
3 both intrinsic and derived types.

4 R616 *type-param-inquiry*            **is** *designator % type-param-name*

5 C620 (R616) The *type-param-name* shall be the name of a type parameter of the declared type of the  
6 object designated by the *designator*.

7 A deferred type parameter of a pointer that is not associated or of an unallocated allocatable variable  
8 shall not be inquired about.

**NOTE 6.9**

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 6.10**

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.

## 9 6.2 Arrays

10 An **array** is a set of scalar data, all of the same type and type parameters, whose individual elements  
11 are arranged in a rectangular pattern. The scalar data that make up an array are the **array elements**.

12 No order of reference to the elements of an array is indicated by the appearance of the array designator,  
13 except where array element ordering (6.2.2.2) is specified.

### 14 6.2.1 Whole arrays

15 A **whole array** is a named array, which may be either a named constant (5.3.12, 5.4.10) or a variable;  
16 no subscript list is appended to the name.

17 The appearance of a whole array variable in an executable construct specifies all the elements of the  
18 array (2.4.5). An assumed-size array is permitted to appear as a whole array in an executable construct

- 1 only as an actual argument in a procedure reference that does not require the shape.  
 2 The appearance of a whole array name in a nonexecutable statement specifies the entire array except  
 3 for the appearance of a whole array name in an equivalence set (5.7.1.4).

#### 4 **6.2.2 Array elements and array sections**

5 R617 *array-element* **is** *data-ref*

6 C621 (R617) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

7 R618 *array-section* **is** *data-ref* [ ( *substring-range* ) ]  
 8 **or** *complex-part-designator*

9 C622 (R618) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a  
 10 *section-subscript-list* with nonzero rank, the *array-section* is a *complex-part-designator* that is  
 11 an array, or another *part-ref* shall have nonzero rank.

12 C623 (R618) If a *substring-range* appears, the rightmost *part-name* shall be of type character.

13 R619 *subscript* **is** *scalar-int-expr*

14 R620 *section-subscript* **is** *subscript*  
 15 **or** *subscript-triplet*  
 16 **or** *vector-subscript*

17 R621 *subscript-triplet* **is** [ *subscript* ] : [ *subscript* ] [ : *stride* ]

18 R622 *stride* **is** *scalar-int-expr*

19 R623 *vector-subscript* **is** *int-expr*

20 C624 (R623) A *vector-subscript* shall be an integer array expression of rank one.

21 C625 (R621) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension  
 22 of an assumed-size array.

23 An array element is a scalar. An array section is an array. If a *substring-range* is present in an *array-*  
 24 *section*, each element is the designated substring of the corresponding element of the array section.

##### **NOTE 6.11**

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 6.12**

Unless otherwise specified, an array element or array section does not have an attribute of the whole array. In particular, an array element or an array section does not have the POINTER or ALLOCATABLE attribute.

##### **NOTE 6.13**

Examples of array elements and array sections are:

```
ARRAY_A(1:N:2)%ARRAY_B(I, J)%STRING(K)(:)    array section
```

## NOTE 6.13 (cont.)

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

1 **6.2.2.1 Array elements**

2 The value of a subscript in an array element shall be within the bounds for that dimension.

3 **6.2.2.2 Array element order**

4 The elements of an array form a sequence known as the **array element order**. The position of an array  
5 element in this sequence is determined by the subscript order value of the subscript list designating the  
6 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 $i = 1, 2, \dots, 15$ .			

7 **6.2.2.3 Array sections**

8 An **array section** is an array subobject optionally followed by a substring range.

9 In an *array-section* having a *section-subscript-list*, each *subscript-triplet* and *vector-subscript* in the  
10 section subscript list indicates a sequence of subscripts, which may be empty. Each subscript in such a  
11 sequence shall be within the bounds for its dimension unless the sequence is empty. The array section is  
12 the set of elements from the array determined by all possible subscript lists obtainable from the single  
13 subscripts or sequences of subscripts specified by each section subscript.

14 In an *array-section* with no *section-subscript-list*, the rank and shape of the array is the rank and shape  
15 of the *part-ref* with nonzero rank; otherwise, the rank of the array section is the number of subscript  
16 triplets and vector subscripts in the section subscript list. The shape is the rank-one array whose  $i$ th  
17 element is the number of integer values in the sequence indicated by the  $i$ th subscript triplet or vector  
18 subscript. If any of these sequences is empty, the array section has size zero. The subscript order of the  
19 elements of an array section is that of the array data object that the array section represents.

### 1 6.2.2.3.1 Subscript triplet

2 A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values.  
 3 The third expression in the subscript triplet is the increment between the subscript values and is called  
 4 the **stride**. The subscripts and stride of a subscript triplet are optional. An omitted first subscript in a  
 5 subscript triplet is equivalent to a subscript whose value is the lower bound for the array and an omitted  
 6 second subscript is equivalent to the upper bound. An omitted stride is equivalent to a stride of 1.

7 The stride shall not be zero.

8 When the stride is positive, the subscripts specified by a triplet form a regularly spaced sequence of  
 9 integers beginning with the first subscript and proceeding in increments of the stride to the largest such  
 10 integer not greater than the second subscript; the sequence is empty if the first subscript is greater than  
 11 the second.

#### NOTE 6.14

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)

12 When the stride is negative, the sequence begins with the first subscript and proceeds in increments of  
 13 the stride down to the smallest such integer equal to or greater than the second subscript; the sequence  
 14 is empty if the second subscript is greater than the first.

#### NOTE 6.15

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 6.16

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.

### 15 6.2.2.3.2 Vector subscript

16 A **vector subscript** designates a sequence of subscripts corresponding to the values of the elements  
 17 of the expression. Each element of the expression shall be defined. A **many-one array section** is an  
 18 array section with a vector subscript having two or more elements with the same value. A many-one  
 19 array section shall appear neither on the left of the equals in an assignment statement nor as an input  
 20 item in a READ statement.

21 An array section with a vector subscript shall not be argument associated with a dummy array that  
 22 is defined or redefined. An array section with a vector subscript shall not be the target in a pointer  
 23 assignment statement. An array section with a vector subscript shall not be an internal file.

#### NOTE 6.17

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:

**NOTE 6.17 (cont.)**

U = (/ 1, 3, 2 /)  
V = (/ 2, 1, 1, 3 /)

Then Z (3, V) consists of elements from the third row of Z in the order:

Z (3, 2)    Z (3, 1)    Z (3, 1)    Z (3, 3)

and Z (U, 2) consists of the column elements:

Z (1, 2)    Z (3, 2)    Z (2, 2)

and Z (U, V) consists of the elements:

Z (1, 2)    Z (1, 1)    Z (1, 1)    Z (1, 3)  
Z (3, 2)    Z (3, 1)    Z (3, 1)    Z (3, 3)  
Z (2, 2)    Z (2, 1)    Z (2, 1)    Z (2, 3)

Because Z (3, V) and Z (U, V) contain duplicate elements from Z, the sections Z (3, V) and Z (U, V) shall not be redefined as sections.

### 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 bounds for its co-dimension. An image selector shall specify an  
7 image index value that is not greater than the number of images.

**NOTE 6.18**

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.

**J3 internal note**

Unresolved Technical Issue 21

This does not appear to be adequately specified. The mapping between co-subscripts (or rather, co-subscript lists) and image indexes is done by handwaving analogy. It is probably not necessary to repeat the formulae, but we need to say exactly how we use those formulae in a more rigorous way. (For a start, “Subscript bounds” clearly does not apply to co-subscripts.)

## 8 6.3 Dynamic association

9 Dynamic control over the allocation, association, and deallocation of pointer targets is provided by  
10 the ALLOCATE, NULLIFY, and DEALLOCATE statements and pointer assignment. ALLOCATE  
11 (6.3.1) creates targets for pointers; pointer assignment (7.4.2) associates pointers with existing targets;  
12 NULLIFY (6.3.2) disassociates pointers from targets, and DEALLOCATE (6.3.3) deallocates targets.

- 1 Dynamic association applies to scalars and arrays of any type.
- 2 The ALLOCATE and DEALLOCATE statements also are used to create and deallocate variables with
- 3 the ALLOCATABLE attribute.

**NOTE 6.19**

Detailed remarks regarding pointers and dynamic association are in C.3.3.

#### 4 **6.3.1 ALLOCATE statement**

- 5 The **ALLOCATE statement** dynamically creates pointer targets and allocatable variables.

6	R626	<i>allocate-stmt</i>	<b>is</b>	ALLOCATE ( [ <i>type-spec</i> :: ] <i>allocation-list</i> ■
7				■ [ <i>alloc-opt-list</i> ] )
8	R627	<i>alloc-opt</i>	<b>is</b>	ERRMSG = <i>errmsg-variable</i>
9			<b>or</b>	MOLD = <i>source-expr</i>
10			<b>or</b>	SOURCE = <i>source-expr</i>
11			<b>or</b>	STAT = <i>stat-variable</i>
12	R628	<i>stat-variable</i>	<b>is</b>	<i>scalar-int-variable</i>
13	R629	<i>errmsg-variable</i>	<b>is</b>	<i>scalar-default-char-variable</i>
14	R630	<i>source-expr</i>	<b>is</b>	<i>expr</i>
15	R631	<i>allocation</i>	<b>is</b>	<i>allocate-object</i> [ ( <i>allocate-shape-spec-list</i> ) ] ■
16				■ [ <i>lbracket allocate-co-array-spec rbracket</i> ]
17	R632	<i>allocate-object</i>	<b>is</b>	<i>variable-name</i>
18			<b>or</b>	<i>structure-component</i>
19	R633	<i>allocate-shape-spec</i>	<b>is</b>	[ <i>lower-bound-expr</i> : ] <i>upper-bound-expr</i>
20	R634	<i>lower-bound-expr</i>	<b>is</b>	<i>scalar-int-expr</i>
21	R635	<i>upper-bound-expr</i>	<b>is</b>	<i>scalar-int-expr</i>
22	R636	<i>allocate-co-array-spec</i>	<b>is</b>	[ <i>allocate-co-shape-spec-list</i> , ] [ <i>lower-bound-expr</i> : ] *
23	R637	<i>allocate-co-shape-spec</i>	<b>is</b>	[ <i>lower-bound-expr</i> : ] <i>upper-bound-expr</i>

- 24 C626 (R632) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.
- 25 C627 (R626) If any *allocate-object* in the statement has a deferred type parameter, either *type-spec* or
- 26 SOURCE= shall appear.
- 27 C628 (R626) If a *type-spec* appears, it shall specify a type with which each *allocate-object* is type
- 28 compatible.
- 29 C629 (R626) If any *allocate-object* is unlimited polymorphic or is of abstract type, either *type-spec* or
- 30 SOURCE= shall appear.
- 31 C630 (R626) A *type-param-value* in a *type-spec* shall be an asterisk if and only if each *allocate-object*
- 32 is a dummy argument for which the corresponding type parameter is assumed.
- 33 C631 (R626) If a *type-spec* appears, the kind type parameter values of each *allocate-object* shall be
- 34 the same as the corresponding type parameter values of the *type-spec*.
- 35 C632 (R631) An *allocate-shape-spec-list* shall appear if and only if the *allocate-object* is an array. An
- 36 *allocate-co-array-spec* shall appear if and only if the *allocate-object* is a co-array.
- 37 C633 (R631) The number of *allocate-shape-specs* in an *allocate-shape-spec-list* shall be the same as the
- 38 rank of the *allocate-object*. The number of *allocate-co-shape-specs* in an *allocate-co-array-spec*

- 1 shall be one less than the co-rank of the *allocate-object*.
- 2 C634 (R627) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.
- 3 C635 (R626) At most one of SOURCE=, MOLD=, and *type-spec* shall appear.
- 4 C636 (R626) Each *allocate-object* shall be type compatible (4.3.1.3) with *source-expr*. If SOURCE=  
5 appears, *source-expr* shall be a scalar or have the same rank as each *allocate-object*.
- 6 C637 (R626) Corresponding kind type parameters of *allocate-object* and *source-expr* shall have the  
7 same values.
- 8 C638 (R632) An *allocate-object* shall not be a co-indexed object.

**NOTE 6.20**

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.
```

- 9 An *allocate-object* or a bound or type parameter of an *allocate-object* shall not depend on the value of  
10 *stat-variable*, the value of *errmsg-variable*, or on the value, bounds, length type parameters, allocation  
11 status, or association status of any *allocate-object* in the same ALLOCATE statement.
- 12 Neither *stat-variable*, *source-expr*, nor *errmsg-variable* shall be allocated within the ALLOCATE state-  
13 ment in which it appears; nor shall they depend on the value, bounds, length type parameters, allocation  
14 status, or association status of any *allocate-object* in the same ALLOCATE statement.
- 15 If *type-spec* is specified, each *allocate-object* is allocated with the specified dynamic type and type pa-  
16 rameter values; if *source-expr* is specified, each *allocate-object* is allocated with the dynamic type and  
17 type parameter values of *source-expr*; otherwise, each *allocate-object* is allocated with its dynamic type  
18 the same as its declared type. The dynamic type shall not have a co-array ultimate component.

**J3 internal note****Unresolved Technical Issue 22**

(minor) It is worth considering how this dynamic type could ever have a co-array subcomponent. In particular, we should constrain this *type-spec* against being such a type.

(major) This is a runtime problem which cannot be avoided by the user in any systematic way. It is a very bad idea to design the language so that “bad things happen” when the user is not in any position to detect that a bad thing might happen.

The easy solution is

- (1) to prohibit type extension from adding co-array components, and
- (2) to constrain the declared type of *source-expr* from having any co-array ultimate components.

Since co-array components are really really special (like, something with a co-array component cannot be pointed to) it does not seem too unreasonable for them not to be includable via type extension.

- 19 If *type-spec* appears and the value of a type parameter it specifies differs from the value of the corre-

- 1 sponding nondeferred type parameter specified in the declaration of any *allocate-object*, an error condition  
 2 occurs. If the value of a nondeferred length type parameter of an *allocate-object* differs from the value  
 3 of the corresponding type parameter of *source-expr*, an error condition occurs.
- 4 If a *type-param-value* in a *type-spec* in an ALLOCATE statement is an asterisk, it denotes the current  
 5 value of that assumed type parameter. If it is an expression, subsequent redefinition or undefinition of  
 6 any entity in the expression does not affect the type parameter value.

**NOTE 6.21**

An example of an ALLOCATE statement is:

```
ALLOCATE (X (N), B (-3 : M, 0:9), STAT = IERR_ALLOC)
```

- 7 When an ALLOCATE statement is executed for an array, the values of the lower bound and upper  
 8 bound expressions determine the bounds of the array. Subsequent redefinition or undefinition of any  
 9 entities in the bound expressions do not affect the array bounds. If the lower bound is omitted, the  
 10 default value is 1. If the upper bound is less than the lower bound, the extent in that dimension is zero  
 11 and the array has zero size.

**NOTE 6.22**

An *allocate-object* may be of type character with zero character length.

- 12 If SOURCE= appears, *source-expr* shall be conformable (2.4.5) with *allocation*. If the value of a non-  
 13 deferred length type parameter of *allocate-object* is different from the value of the corresponding type  
 14 parameter of *source-expr*, an error condition occurs. If the allocation is successful, the value of *allocate-*  
 15 *object* becomes that of *source-expr*.
- 16 If MOLD= appears and *source-expr* is a variable, its value need not be defined.

**NOTE 6.23**

The *source-expr* can be an array or scalar independently of whether an *allocate-object* is an array or a scalar. For MOLD=*source-expr*, only the dynamic type and type parameter values of *source-expr* are relevant; its rank, shape, and value are not.

**NOTE 6.24**

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
! ...
ALLOCATE (NEW, SOURCE=OLD) ! Allocate NEW with the value and dynamic type of OLD
```

A more extensive example is given in C.3.2.

- 17 If the STAT= specifier appears, successful execution of the ALLOCATE statement causes the *stat-*  
 18 *variable* to become defined with a value of zero. If an error condition occurs during the execution  
 19 of the ALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive  
 20 integer value and each *allocate-object* will have a processor-dependent status; each *allocate-object* that  
 21 was successfully allocated shall have an allocation status of allocated or a pointer association status of  
 22 associated; each *allocate-object* that was not successfully allocated shall retain its previous allocation  
 23 status or pointer association status.



1 If an error condition occurs during execution of an ALLOCATE statement that does not contain the  
 2 STAT= specifier, execution of the program is terminated.

3 The ERRMSG= specifier is described in 6.3.1.3.

#### 4 **6.3.1.1 Allocation of allocatable variables**

5 The allocation status of an allocatable entity is one of the following at any time during the execution of  
 6 a program.

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 A saved allocatable object has an initial status of unallocated. The status may change during the  
 22 execution of the program.

23 When the allocation status of an allocatable variable changes, the allocation status of any associated  
 24 allocatable variable changes accordingly. Allocation of an allocatable variable establishes values for the  
 25 deferred type parameters of all associated allocatable variables.

26 An unsaved allocatable object that is a local variable of a procedure has a status of unallocated at the  
 27 beginning of each invocation of the procedure. The status may change during execution of the procedure.  
 28 An unsaved allocatable object that is a local variable of a module or submodule, or a subobject thereof,  
 29 has an initial status of unallocated. The status may change during execution of the program.

30 When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate  
 31 components have an allocation status of unallocated.

32 The value of each *lower-bound-expr* and each *upper-bound-expr* in an *allocate-co-array-spec* shall be the  
 33 same on each image.

34 If an *allocation* specifies a co-array, its dynamic type and the values of each length type parameter shall  
 35 be the same on each image.

36 There is implicit synchronization of all images in association with each ALLOCATE statement that  
 37 involves one or more co-arrays. On each image, execution of the segment (8.5.1) following the statement  
 38 is delayed until all other images have executed the same statement the same number of times.

#### **NOTE 6.25**

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

**NOTE 6.25 (cont.)**

for detecting or resolving deadlock problems (such as two images waiting on different ALLOCATE statements).

**1 6.3.1.2 Allocation of pointer targets**

2 Allocation of a pointer creates an object that implicitly has the TARGET attribute. Following successful  
 3 execution of an ALLOCATE statement for a pointer, the pointer is associated with the target and may  
 4 be used to reference or define the target. Additional pointers may become associated with the pointer  
 5 target or a part of the pointer target by pointer assignment. It is not an error to allocate a pointer  
 6 that is already associated with a target. In this case, a new pointer target is created as required by the  
 7 attributes of the pointer and any array bounds, type, and type parameters specified by the ALLOCATE  
 8 statement. The pointer is then associated with this new target. Any previous association of the pointer  
 9 with a target is broken. If the previous target had been created by allocation, it becomes inaccessible  
 10 unless other pointers are associated with it. The ASSOCIATED intrinsic function (13.7.15) may be used  
 11 to determine whether a pointer that does not have undefined association status is associated.

12 At the beginning of execution of a function whose result is a pointer, the association status of the result  
 13 pointer is undefined. Before such a function returns, it shall either associate a target with this pointer  
 14 or cause the association status of this pointer to become defined as disassociated.

**15 6.3.1.3 ERRMSG= specifier**

16 If an error condition occurs during execution of an ALLOCATE or DEALLOCATE statement, the  
 17 processor shall assign an explanatory message to *errmsg-variable*. If no such condition occurs, the  
 18 processor shall not change the value of *errmsg-variable*.

**19 6.3.2 NULLIFY statement**

20 The **NULLIFY statement** causes pointers to be disassociated.

21 R638	<i>nullify-stmt</i>	<b>is</b> NULLIFY ( <i>pointer-object-list</i> )
22 R639	<i>pointer-object</i>	<b>is</b> <i>variable-name</i>
23		<b>or</b> <i>structure-component</i>
24		<b>or</b> <i>proc-pointer-name</i>

25 C639 (R639) Each *pointer-object* shall have the POINTER attribute.

26 A *pointer-object* shall not depend on the value, bounds, or association status of another *pointer-object*  
 27 in the same NULLIFY statement.

**NOTE 6.26**

When a NULLIFY statement is applied to a polymorphic pointer (4.3.1.3), its dynamic type becomes the declared type.

**28 6.3.3 DEALLOCATE statement**

29 The **DEALLOCATE statement** causes allocatable variables to be deallocated; it causes pointer tar-  
 30 gets to be deallocated and the pointers to be disassociated.

31 R640	<i>deallocate-stmt</i>	<b>is</b> DEALLOCATE ( <i>allocate-object-list</i> [ , <i>dealloc-opt-list</i> ] )
---------	------------------------	------------------------------------------------------------------------------------

32 C640 (R640) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

1 R641 *dealloc-opt*                            **is** STAT = *stat-variable*  
 2                                                    **or** ERRMSG = *errmsg-variable*

3 C641 (R641) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

4 An *allocate-object* shall not depend on the value, bounds, allocation status, or association status of  
 5 another *allocate-object* in the same DEALLOCATE statement; it also shall not depend on the value of  
 6 the *stat-variable* or *errmsg-variable* in the same DEALLOCATE statement.

7 Neither *stat-variable* nor *errmsg-variable* shall be deallocated within the same DEALLOCATE state-  
 8 ment; they also shall not depend on the value, bounds, allocation status, or association status of any  
 9 *allocate-object* in the same DEALLOCATE statement.

10 If the STAT= specifier appears, successful execution of the DEALLOCATE statement causes the *stat-*  
 11 *variable* to become defined with a value of zero. If an error condition occurs during the execution of  
 12 the DEALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive  
 13 integer value and each *allocate-object* that was successfully deallocated shall have an allocation status of  
 14 unallocated or a pointer association status of disassociated. Each *allocate-object* that was not successfully  
 15 deallocated shall retain its previous allocation status or pointer association status.

#### NOTE 6.27

The status of objects that were not successfully deallocated can be individually checked with the ALLOCATED or ASSOCIATED intrinsic functions.

16 If an error condition occurs during execution of a DEALLOCATE statement that does not contain the  
 17 STAT= specifier, execution of the program is terminated.

18 The ERRMSG= specifier is described in 6.3.1.3.

#### NOTE 6.28

An example of a DEALLOCATE statement is:

```
DEALLOCATE (X, B)
```

### 19 6.3.3.1 Deallocation of allocatable variables

20 Deallocating an unallocated allocatable variable causes an error condition in the DEALLOCATE state-  
 21 ment. Deallocating an allocatable variable with the TARGET attribute causes the pointer association  
 22 status of any pointer associated with it to become undefined.

23 When the execution of a procedure is terminated by execution of a RETURN or END statement, an  
 24 allocatable variable that is a named local variable of the procedure retains its allocation and definition  
 25 status if it has the SAVE attribute or is a function result variable or a subobject thereof; otherwise, it  
 26 is deallocated.

27 When a BLOCK construct terminates, an allocatable variable that is a named local variable of the con-  
 28 struct retains its allocation and definition status if it has the SAVE attribute; otherwise it is deallocated.

#### NOTE 6.29

The ALLOCATED intrinsic function may be used to determine whether a variable is allocated or unallocated.

29 If an unsaved allocatable object is a local variable of a module or submodule, and it is allocated when  
 30 execution of a RETURN or END statement results in no active scoping unit referencing the module or

- 1 submodule, it is processor-dependent whether the object retains its allocation status or is deallocated.

#### NOTE 6.30

The following example illustrates the effects of SAVE on allocation status.

```

MODULE MOD1
  TYPE INITIALIZED_TYPE
    INTEGER :: I = 1 ! Default initialization
  END TYPE INITIALIZED_TYPE
  SAVE :: SAVED1, SAVED2
  INTEGER :: SAVED1, UNSAVED1
  TYPE(INITIALIZED_TYPE) :: SAVED2, UNSAVED2
  ALLOCATABLE :: SAVED1(:), SAVED2(:), UNSAVED1(:), UNSAVED2(:)
END MODULE MOD1
PROGRAM MAIN
  CALL SUB1 ! The values returned by the ALLOCATED intrinsic calls
            ! in the PRINT statement are:
            ! .FALSE., .FALSE., .FALSE., and .FALSE.
            ! Module MOD1 is used, and its variables are allocated.
            ! After return from the subroutine, whether the variables
            ! which were not specified with the SAVE attribute
            ! retain their allocation status is processor dependent.
  CALL SUB1 ! The values returned by the first two ALLOCATED intrinsic
            ! calls in the PRINT statement are:
            ! .TRUE., .TRUE.
            ! The values returned by the second two ALLOCATED
            ! intrinsic calls in the PRINT statement are
            ! processor dependent and each could be either
            ! .TRUE. or .FALSE.

  CONTAINS
    SUBROUTINE SUB1
      USE MOD1 ! Brings in saved and unsaved variables.
      PRINT *, ALLOCATED(SAVED1), ALLOCATED(SAVED2), &
        ALLOCATED(UNSAVED1), ALLOCATED(UNSAVED2)
      IF (.NOT. ALLOCATED(SAVED1)) ALLOCATE(SAVED1(10))
      IF (.NOT. ALLOCATED(SAVED2)) ALLOCATE(SAVED2(10))
      IF (.NOT. ALLOCATED(UNSAVED1)) ALLOCATE(UNSAVED1(10))
      IF (.NOT. ALLOCATED(UNSAVED2)) ALLOCATE(UNSAVED2(10))
    END SUBROUTINE SUB1
END PROGRAM MAIN

```

- 2 If an executable construct references a function whose result is either allocatable or a structure with  
 3 a subobject that is allocatable, and the function reference is executed, an allocatable result and any  
 4 subobject that is an allocated allocatable entity in the result returned by the function is deallocated  
 5 after execution of the innermost executable construct containing the reference.
- 6 If a specification expression in a scoping unit references a function whose result is either allocatable or  
 7 a structure with a subobject that is allocatable, and the function reference is executed, an allocatable  
 8 result and any subobject that is an allocated allocatable entity in the result returned by the function is  
 9 deallocated before execution of the first executable statement in the scoping unit.
- 10 When a procedure is invoked, any allocated allocatable object that is an actual argument associated with  
 11 an INTENT(OUT) allocatable dummy argument is deallocated; any allocated allocatable object that is  
 12 a subobject of an actual argument associated with an INTENT(OUT) dummy argument is deallocated.

- 1 When an intrinsic assignment statement (7.4.1.3) is executed, any allocated allocatable subobject of the
- 2 variable is deallocated before the assignment takes place.
- 3 When a variable of derived type is deallocated, any allocated allocatable subobject is deallocated.
- 4 If an allocatable component is a subobject of a finalizable object, that object is finalized before the
- 5 component is automatically deallocated.
- 6 The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a
- 7 *dealloc-opt-list*.

**NOTE 6.31**

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. On the next invocation of PROCESS, TEMP will have an allocation status of unallocated.

- 8 There is implicit synchronization of all images in association with each DEALLOCATE statement that
- 9 involves one or more co-arrays. On each image, execution of the segment (8.5.1) following the statement
- 10 is delayed until all other images have executed the same statement the same number of times.
- 11 There is also an implicit synchronization of all images in association with the deallocation of a co-array
- 12 or co-array subcomponent caused by the execution of a RETURN or END statement or the termination
- 13 of a BLOCK construct.
- 14 **6.3.3.2 Deallocation of pointer targets**
- 15 If a pointer appears in a DEALLOCATE statement, its association status shall be defined. Deallocating
- 16 a pointer that is disassociated or whose target was not created by an ALLOCATE statement causes an
- 17 error condition in the DEALLOCATE statement. If a pointer is associated with an allocatable entity,
- 18 the pointer shall not be deallocated.
- 19 If a pointer appears in a DEALLOCATE statement, it shall be associated with the whole of an object
- 20 that was created by allocation. Deallocating a pointer target causes the pointer association status of
- 21 any other pointer that is associated with the target or a portion of the target to become undefined.



## 7 Expressions and assignment

This clause describes the formation, interpretation, and evaluation rules for expressions, intrinsic and defined assignment, pointer assignment, masked array assignment (WHERE), and FORALL.

### 7.1 Expressions

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 or defined (7.2). 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 (7.1.4).

#### 7.1.1 Form of an expression

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*. The rules given below specify the syntax of an expression. The semantics are specified in 7.2.

##### 7.1.1.1 Primary

R701	<i>primary</i>	<b>is</b>	<i>constant</i>
		<b>or</b>	<i>designator</i>
		<b>or</b>	<i>array-constructor</i>
		<b>or</b>	<i>structure-constructor</i>
		<b>or</b>	<i>function-reference</i>
		<b>or</b>	<i>type-param-inquiry</i>
		<b>or</b>	<i>type-param-name</i>
		<b>or</b>	<i>( expr )</i>

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> )
-----------------	-------------------------------------------

**1 7.1.1.2 Level-1 expressions**

2 Defined unary operators have the highest operator precedence (Table 7.9). Level-1 expressions are  
3 primaries optionally operated on by defined unary operators:

4 R702 *level-1-expr* **is** [ *defined-unary-op* ] *primary*

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

6 C703 (R703) A *defined-unary-op* shall not contain more than 63 letters and shall not be the same as  
7 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)

**8 7.1.1.3 Level-2 expressions**

9 Level-2 expressions are level-1 expressions optionally involving the numeric operators *power-op*, *mult-op*,  
10 and *add-op*.

11 R704 *mult-operand* **is** *level-1-expr* [ *power-op mult-operand* ]

12 R705 *add-operand* **is** [ *add-operand mult-op* ] *mult-operand*

13 R706 *level-2-expr* **is** [ [ *level-2-expr* ] *add-op* ] *add-operand*

14 R707 *power-op* **is** \*\*

15 R708 *mult-op* **is** \*

16 **or** /

17 R709 *add-op* **is** +

18 **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)
+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)



**NOTE 7.3 (cont.)**

A more complicated example of a level-2 expression is:

- A + D \* E + B \*\* C

**1 7.1.1.4 Level-3 expressions**

2 Level-3 expressions are level-2 expressions optionally involving the character operator and bits concatenate 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 7.1.1.5 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:

<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

### 1 7.1.1.6 Level-5 expressions

2 Level-5 expressions are level-4 expressions optionally involving the logical and bits operators *not-op*,  
3 *and-op*, *or-op*, and *equiv-op*.

4 R714	<i>and-operand</i>	<b>is</b>	[ <i>not-op</i> ] <i>level-4-expr</i>
5 R715	<i>or-operand</i>	<b>is</b>	[ <i>or-operand and-op</i> ] <i>and-operand</i>
6 R716	<i>equiv-operand</i>	<b>is</b>	[ <i>equiv-operand or-op</i> ] <i>or-operand</i>
7 R717	<i>level-5-expr</i>	<b>is</b>	[ <i>level-5-expr equiv-op</i> ] <i>equiv-operand</i>
8 R718	<i>not-op</i>	<b>is</b>	.NOT.
9 R719	<i>and-op</i>	<b>is</b>	.AND.
10 R720	<i>or-op</i>	<b>is</b>	.OR.
11 R721	<i>equiv-op</i>	<b>is</b>	.EQV.
12		<b>or</b>	.NEQV.
13		<b>or</b>	.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

### 14 7.1.1.7 General form of an expression

15 Expressions are level-5 expressions optionally involving defined binary operators. Defined binary oper-  
16 ators have the lowest operator precedence (Table 7.9).

17 R722	<i>expr</i>	<b>is</b>	[ <i>expr defined-binary-op</i> ] <i>level-5-expr</i>
18 R723	<i>defined-binary-op</i>	<b>is</b>	. <i>letter</i> [ <i>letter</i> ] ... .

19 C704 (R723) A *defined-binary-op* shall not contain more than 63 letters and shall not be the same as  
20 any *intrinsic-operator* or *logical-literal-constant*.

#### NOTE 7.7

Simple examples of an expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-5-expr</i> (R717)
B.UNION.C	<i>expr</i> (R722)

More complicated examples of an expression are:

**NOTE 7.7 (cont.)**

```

(B .INTERSECT. C) .UNION. (X - Y)
A + B == C * D
.INVERSE. (A + B)
A + B .AND. C * D
E // G == H (1:10)

```

**7.1.2 Intrinsic operations**

- 2 An **intrinsic operation** is either an intrinsic unary operation or an intrinsic binary operation. An  
3 **intrinsic unary operation** is an operation of the form *intrinsic-operator*  $x_2$  where  $x_2$  is of an intrinsic  
4 type (4.4) listed in Table 7.1 for the unary intrinsic operator.
- 5 An **intrinsic binary operation** is an operation of the form  $x_1$  *intrinsic-operator*  $x_2$  where  $x_1$  and  
6  $x_2$  are of the intrinsic types (4.4) listed in Table 7.1 for the binary intrinsic operator and are in shape  
7 conformance (7.1.5).

**Table 7.1: Type of operands and results for intrinsic operators**

Intrinsic operator <i>op</i>	Type of $x_1$	Type of $x_2$	Type of $[x_1] \text{ } op \text{ } x_2$
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
	C	C	L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R	L, L
	R	I, R	L, L
	C	C	L
.NOT.		L, B	L, B
.AND., .OR., .EQV., .NEQV.	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. For the intrinsic operators with operands of type character, the kind type parameters of the operands shall be the same.			

- 8 A **numeric intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is a numeric  
9 operator (+, −, \*, /, or \*\*). A **numeric intrinsic operator** is the operator in a numeric intrinsic  
10 operation.

- 11 For numeric intrinsic binary operations, the two operands may be of different numeric types or different  
12 kind type parameters. Except for a value raised to an integer power, if the operands have different types  
13 or kind type parameters, the effect is as if each operand that differs in type or kind type parameter from  
14 those of the result is converted to the type and kind type parameter of the result before the operation  
15 is performed. When a value of type real or complex is raised to an integer power, the integer operand

1 need not be converted.

2 The **character intrinsic operation** is the intrinsic operation for which the *intrinsic-operator* is `(//)`  
 3 and both operands are of type character. The operands shall have the same kind type parameter. The  
 4 **character intrinsic operator** is the operator in a character intrinsic operation.

5 A **logical intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is `.AND.`, `.OR.`,  
 6 `.XOR.`, `.NOT.`, `.EQV.`, or `.NEQV.` and both operands are of type logical. A **logical intrinsic operator**  
 7 is the operator in a logical intrinsic operation.

8 A **bits intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is `//`, `.AND.`, `.OR.`,  
 9 `.XOR.`, `.NOT.`, `.EQV.`, or `.NEQV.` and at least one operand is of type bits. A **bits intrinsic operator**  
 10 is the operator in a bits intrinsic operation.

11 For bits intrinsic operations other than concatenation `(//)`, the two operands may be of different types  
 12 or different kind type parameters. The effect is as if each operand that differs in type or kind type  
 13 parameter from those of the result is converted to the type and kind type parameter of the result before  
 14 the operation is performed.

15 A **relational intrinsic operator** is an *intrinsic-operator* that is `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, `.LE.`,  
 16 `==`, `/=`, `>`, `>=`, `<`, or `<=`. A **relational intrinsic operation** is an intrinsic operation for which the  
 17 *intrinsic-operator* is a relational intrinsic operator. A **numeric relational intrinsic operation** is a  
 18 relational intrinsic operation for which both operands are of numeric type. A **character relational**  
 19 **intrinsic operation** is a relational intrinsic operation for which both operands are of type character.  
 20 The kind type parameters of the operands of a character relational intrinsic operation shall be the same.  
 21 A **bits relational intrinsic operation** is a relational intrinsic operation for which at least one of the  
 22 operands is of type bits.

23 If both operands of a bits relational operation do not have the same kind type parameter, the operand  
 24 with the smaller kind type parameter is converted to the same kind as the other operand. If one operand  
 25 of a bits relational operation is not of type bits, it is converted to type bits with the same kind type  
 26 parameter as the other operand. Any conversion takes place before the operation is evaluated.

### 27 7.1.3 Defined operations

28 A **defined operation** is either a defined unary operation or a defined binary operation. A **defined**  
 29 **unary operation** is an operation that has the form *defined-unary-op*  $x_2$  or *intrinsic-operator*  $x_2$  and  
 30 that is defined by a function and a generic interface (4.5.2, 12.4.2.1).

31 A function defines the unary operation *op*  $x_2$  if

- 32 (1) the function is specified with a FUNCTION (12.6.2.1) or ENTRY (12.6.2.5) statement that  
 33 specifies one dummy argument  $d_2$ ,
- 34 (2) either
  - 35 (a) a generic interface (12.4.2.1) provides the function with a *generic-spec* of OPERA-  
 36 TOR (*op*), or
  - 37 (b) there is a generic binding (4.5.2) in the declared type of  $x_2$  with a *generic-spec* of  
 38 OPERATOR (*op*) and there is a corresponding binding to the function in the dynamic  
 39 type of  $x_2$ ,
- 40 (3) the type of  $d_2$  is compatible with the dynamic type of  $x_2$ ,
- 41 (4) the type parameters, if any, of  $d_2$  match the corresponding type parameters of  $x_2$ , and
- 42 (5) either
  - 43 (a) the rank of  $x_2$  matches that of  $d_2$  or
  - 44 (b) the function is elemental and there is no other function that defines the operation.

- 1 If  $d_2$  is an array, the shape of  $x_2$  shall match the shape of  $d_2$ .
- 2 A **defined binary operation** is an operation that has the form  $x_1$  *defined-binary-op*  $x_2$  or  $x_1$  *intrinsic-*  
 3 *operator*  $x_2$  and that is defined by a function and a generic interface.
- 4 A function defines the binary operation  $x_1$  *op*  $x_2$  if
- 5 (1) the function is specified with a FUNCTION (12.6.2.1) or ENTRY (12.6.2.5) statement that  
 6 specifies two dummy arguments,  $d_1$  and  $d_2$ ,
  - 7 (2) either
    - 8 (a) a generic interface (12.4.2.1) provides the function with a *generic-spec* of OPERA-  
 9 TOR (*op*), or
    - 10 (b) there is a generic binding (4.5.2) in the declared type of  $x_1$  or  $x_2$  with a *generic-*  
 11 *spec* of OPERATOR (*op*) and there is a corresponding binding to the function in the  
 12 dynamic type of  $x_1$  or  $x_2$ , respectively,
  - 13 (3) the types of  $d_1$  and  $d_2$  are compatible with the dynamic types of  $x_1$  and  $x_2$ , respectively,
  - 14 (4) the type parameters, if any, of  $d_1$  and  $d_2$  match the corresponding type parameters of  $x_1$   
 15 and  $x_2$ , respectively, and
  - 16 (5) either
    - 17 (a) the ranks of  $x_1$  and  $x_2$  match those of  $d_1$  and  $d_2$  or
    - 18 (b) the function is elemental,  $x_1$  and  $x_2$  are conformable, and there is no other function  
 19 that defines the operation.
- 20 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 7.8**

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.

- 21 An **extension operation** is a defined operation in which the operator is of the form *defined-unary-op*  
 22 or *defined-binary-op*. Such an operator is called an **extension operator**. The operator used in an  
 23 extension operation may be such that a generic interface for the operator may specify more than one  
 24 function.
- 25 A **defined elemental operation** is a defined operation for which the function is elemental (12.8).

**7.1.4 Type, type parameters, and shape of an expression**

- 27 The type, type parameters, and shape of an expression depend on the operators and on the types, type  
 28 parameters, and shapes of the primaries used in the expression, and are determined recursively from  
 29 the syntactic form of the expression. The type of an expression is one of the intrinsic types (4.4) or a  
 30 derived type (4.5).
- 31 If an expression is a polymorphic primary or defined operation, the type parameters and the declared and  
 32 dynamic types of the expression are the same as those of the primary or defined operation. Otherwise  
 33 the type parameters and dynamic type of the expression are the same as its declared type and type  
 34 parameters; they are referred to simply as the type and type parameters of the expression.
- 35 R724 *logical-expr* **is** *expr*
- 36 C705 (R724) *logical-expr* shall be of type logical.

- 1 R725 *char-expr* **is** *expr*
- 2 C706 (R725) *char-expr* shall be of type character.
- 3 R726 *default-char-expr* **is** *expr*
- 4 C707 (R726) *default-char-expr* shall be of type default character.
- 5 R727 *int-expr* **is** *expr*
- 6 C708 (R727) *int-expr* shall be of type integer.
- 7 R728 *numeric-expr* **is** *expr*
- 8 C709 (R728) *numeric-expr* shall be of type integer, real, or complex.

#### 9 7.1.4.1 Type, type parameters, and shape of a primary

10 The type, type parameters, and shape of a primary are determined according to whether the primary is a  
 11 constant, variable, array constructor, structure constructor, function reference, type parameter inquiry,  
 12 type parameter name, or parenthesized expression. If a primary is a constant, its type, type parameters,  
 13 and shape are those of the constant. If it is a structure constructor, it is scalar and its type and type  
 14 parameters are as described in 4.5.10. If it is an array constructor, its type, type parameters, and shape  
 15 are as described in 4.7. If it is a variable or function reference, its type, type parameters, and shape are  
 16 those of the variable (5.2, 5.3) or the function reference (12.5.2), respectively. If the function reference  
 17 is generic (12.4.2.1, 13.5) then its type, type parameters, and shape are those of the specific function  
 18 referenced, which is determined by the types, type parameters, and ranks of its actual arguments as  
 19 specified in 12.5.4.1. If it is a type parameter inquiry or type parameter name, it is a scalar integer with  
 20 the kind of the type parameter.

21 If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expres-  
 22 sion.

23 The associated target object is referenced if a pointer appears as

- 24 (1) a primary in an intrinsic or defined operation,  
 25 (2) the *expr* of a parenthesized primary, or  
 26 (3) the only primary on the right-hand side of an intrinsic assignment statement.

27 The type, type parameters, and shape of the primary are those of the current target. If the pointer is  
 28 not associated with a target, it may appear as a primary only as an actual argument in a reference to  
 29 a procedure whose corresponding dummy argument is declared to be a pointer, or as the target in a  
 30 pointer assignment statement.

31 A disassociated array pointer or an unallocated allocatable array has no shape but does have rank.  
 32 The type, type parameters, and rank of the result of the NULL intrinsic function depend on context  
 33 (13.7.131).

#### 34 7.1.4.2 Type, type parameters, and shape of the result of an operation

35 The type of the result of an intrinsic operation  $[x_1] \text{ op } x_2$  is specified by Table 7.1. The shape of the  
 36 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  $x_1$   
 37 otherwise.

38 The type, type parameters, and shape of the result of a defined operation  $[x_1] \text{ op } x_2$  are specified by the  
 39 function defining the operation (7.2).

40 An expression of an intrinsic type has a kind type parameter. An expression of type character also has

1 a character length parameter.

2 The type parameters of the result of an intrinsic operation are as follows.

- 3 (1) For an expression  $x_1 // x_2$  where  $//$  is the character intrinsic operator and  $x_1$  and  $x_2$  are  
4 of type character, the character length parameter is the sum of the lengths of the operands  
5 and the kind type parameter is the kind type parameter of  $x_1$ , which shall be the same as  
6 the kind type parameter of  $x_2$ .
- 7 (2) For an expression  $op x_2$  where  $op$  is an intrinsic unary operator and  $x_2$  is of type integer,  
8 real, complex, logical, or bits, the kind type parameter of the expression is that of the  
9 operand.
- 10 (3) For an expression  $x_1 op x_2$  where  $op$  is a numeric intrinsic binary operator with one operand  
11 of type integer and the other of type real or complex, the kind type parameter of the  
12 expression is that of the real or complex operand.
- 13 (4) For an expression  $x_1 op x_2$  where  $op$  is a numeric intrinsic binary operator with both  
14 operands of the same type and kind type parameters, or with one real and one complex  
15 with the same kind type parameters, the kind type parameter of the expression is identical  
16 to that of each operand. In the case where both operands are integer with different kind type  
17 parameters, the kind type parameter of the expression is that of the operand with the greater  
18 decimal exponent range if the decimal exponent ranges are different; if the decimal exponent  
19 ranges are the same, the kind type parameter of the expression is processor dependent, but  
20 it is the same as that of one of the operands. In the case where both operands are any  
21 of type real or complex with different kind type parameters, the kind type parameter of  
22 the expression is that of the operand with the greater decimal precision if the decimal  
23 precisions are different; if the decimal precisions are the same, the kind type parameter of  
24 the expression is processor dependent, but it is the same as that of one of the operands.
- 25 (5) For an expression  $x_1 op x_2$  where  $op$  is a logical intrinsic binary operator with both operands  
26 of the same kind type parameter, the kind type parameter of the expression is identical to  
27 that of each operand. In the case where both operands are of type logical with different  
28 kind type parameters, the kind type parameter of the expression is processor dependent,  
29 but it is the same as that of one of the operands.
- 30 (6) For an expression  $x_1 // x_2$  where both operands are of type bits, the kind type parameter  
31 of the result is the sum of the kind type parameters of the operands.
- 32 (7) For an expression  $x_1 op x_2$  where  $op$  is a bits intrinsic binary operator other than  $//$ , the  
33 type of the expression is bits and the kind type parameter of the expression is the maximum  
34 of the kind type parameters of  $x_1$  and  $x_2$ .
- 35 (8) For an expression  $x_1 op x_2$  where  $op$  is a relational intrinsic operator, the expression has  
36 the default logical kind type parameter.

37 C710 The kind type parameter for the result of a bits concatenation operation expression shall be a  
38 bits kind type parameter value supported by the processor.

### 39 7.1.5 Conformability rules for elemental operations

40 An **elemental operation** is an intrinsic operation or a defined elemental operation. Two entities are  
41 in **shape conformance** if both are arrays of the same shape, or one or both are scalars.

42 For all elemental binary operations, the two operands shall be in shape conformance. In the case where  
43 one is a scalar and the other an array, the scalar is treated as if it were an array of the same shape as  
44 the array operand with every element, if any, of the array equal to the value of the scalar.

### 1 7.1.6 Specification expression

2 A **specification expression** is an expression with limitations that make it suitable for use in speci-  
 3 fications such as length type parameters (C404) and array bounds (R513, R514). A *specification-expr*  
 4 shall be an initialization expression unless it is in an interface body (12.4.2.1), the specification part of  
 5 a subprogram, or the *declaration-type-spec* of a FUNCTION statement (12.6.2.1).

6 R729 *specification-expr*                      **is** *scalar-int-expr*

7 C711 (R729) The *scalar-int-expr* shall be a restricted expression.

8 A **restricted expression** is an expression in which each operation is intrinsic and each primary is

- 9 (1) a constant or subobject of a constant,
- 10 (2) an object designator with a base object that is a dummy argument that has neither the  
 11 OPTIONAL nor the INTENT (OUT) attribute,
- 12 (3) an object designator with a base object that is in a common block,
- 13 (4) an object designator with a base object that is made accessible by use association or host  
 14 association,
- 15 (5) an array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-*  
 16 *control* is a restricted expression,
- 17 (6) a structure constructor where each component is a restricted expression,
- 18 (7) a specification inquiry where each designator or function argument is
  - 19 (a) a restricted expression or
  - 20 (b) a variable whose properties inquired about are not
    - 21 (i) dependent on the upper bound of the last dimension of an assumed-size array,
    - 22 (ii) deferred, or
    - 23 (iii) defined by an expression that is not a restricted expression,
- 24 (8) a reference to any other standard intrinsic function where each argument is a restricted  
 25 expression,
- 26 (9) a reference to a specification function where each argument is a restricted expression,
- 27 (10) a type parameter of the derived type being defined,
- 28 (11) an *ac-do-variable* within an array constructor where each *scalar-int-expr* of the correspond-  
 29 ing *ac-implied-do-control* is a restricted expression, or
- 30 (12) a restricted expression enclosed in parentheses,

31 where each subscript, section subscript, substring starting point, substring ending point, and type pa-  
 32 rameter value is a restricted expression, and where any final subroutine that is invoked is pure.

33 A **specification inquiry** is a reference to

- 34 (1) an array inquiry function (13.5.7),
- 35 (2) the bit inquiry function BIT\_SIZE,
- 36 (3) the character inquiry function LEN,
- 37 (4) the kind inquiry function KIND,
- 38 (5) the bits kind inquiry function BITS\_KIND,
- 39 (6) the character inquiry function NEW\_LINE,
- 40 (7) a numeric inquiry function (13.5.6),
- 41 (8) a type parameter inquiry (6.1.4),
- 42 (9) an IEEE inquiry function (14.9.1),
- 43 (10) the function C.SIZEOF from the intrinsic module ISO\_C\_BINDING (15.2.3.6) , or



1 (11) the `COMPILER_VERSION` or `COMPILER_OPTIONS` inquiry functions from the intrinsic  
2 module `ISO_FORTRAN_ENV` (13.8.3.3, 13.8.3.4).

3 A function is a **specification function** if it is a pure function, is not a standard intrinsic function, is  
4 not an internal function, is not a statement function, and does not have a dummy procedure argument.

5 Evaluation of a specification expression shall not directly or indirectly cause a procedure defined by the  
6 subprogram in which it appears to be invoked.

#### NOTE 7.9

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.

7 A variable in a specification expression shall have its type and type parameters, if any, specified by a  
8 previous declaration in the same scoping unit, by the implicit typing rules in effect for the scoping unit,  
9 or by host or use association. If a variable in a specification expression is typed by the implicit typing  
10 rules, its appearance in any subsequent type declaration statement shall confirm the implied type and  
11 type parameters.

12 If a specification expression includes a specification inquiry that depends on a type parameter or an  
13 array bound of an entity specified in the same *specification-part*, the type parameter or array bound  
14 shall be specified in a prior specification of the *specification-part*. The prior specification may be to the  
15 left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*. If a  
16 specification expression includes a reference to the value of an element of an array specified in the same  
17 *specification-part*, the array shall be completely specified in prior declarations.

#### NOTE 7.10

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
```

### 18 7.1.7 Initialization expression

19 An **initialization expression** is an expression with limitations that make it suitable for use as a kind  
20 type parameter, initializer, or named constant. It is an expression in which each operation is intrinsic,  
21 and each primary is

- 22 (1) a constant or subobject of a constant,
- 23 (2) an array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-control* is an initialization expression,
- 24 (3) a structure constructor where each *component-spec* corresponding to
  - 26 (a) an allocatable component is a reference to the intrinsic function `NULL`,
  - 27 (b) a pointer component is a reference to the intrinsic function `NULL` or an initialization target, and
  - 28 (c) any other component is an initialization expression,

- 1 (4) a reference to an elemental standard intrinsic function, where each argument is an initial-  
2 ization expression,
- 3 (5) a reference to a transformational standard intrinsic function other than NULL, where each  
4 argument is an initialization expression,
- 5 (6) A reference to the transformational intrinsic function NULL that does not have an argu-  
6 ment with a type parameter that is assumed or is defined by an expression that is not an  
7 initialization expression,
- 8 (7) a reference to the transformational function IEEE\_SELECTED\_REAL\_KIND from the in-  
9 trinsic module IEEE\_ARITHMETIC (14), where each argument is an initialization expres-  
10 sion.
- 11 (8) a specification inquiry where each designator or function argument is
  - 12 (a) an initialization expression or
  - 13 (b) a variable whose properties inquired about are not
    - 14 (i) assumed,
    - 15 (ii) deferred, or
    - 16 (iii) defined by an expression that is not an initialization expression,
- 17 (9) a kind type parameter of the derived type being defined,
- 18 (10) a *data-i-do-variable* within a *data-implied-do*,
- 19 (11) an *ac-do-variable* within an array constructor where each *scalar-int-expr* of the correspond-  
20 ing *ac-implied-do-control* is an initialization expression, or
- 21 (12) an initialization expression enclosed in parentheses,

22 and where each subscript, section subscript, substring starting point, substring ending point, and type  
23 parameter value is an initialization expression.

24 R730 *initialization-expr* **is** *expr*

25 C712 (R730) *initialization-expr* shall be an initialization expression.

26 R731 *char-initialization-expr* **is** *char-expr*

27 C713 (R731) *char-initialization-expr* shall be an initialization expression.

28 R732 *int-initialization-expr* **is** *int-expr*

29 C714 (R732) *int-initialization-expr* shall be an initialization expression.

30 R733 *logical-initialization-expr* **is** *logical-expr*

31 C715 (R733) *logical-initialization-expr* shall be an initialization expression.

32 If an initialization expression includes a specification inquiry that depends on a type parameter or an  
33 array bound of an entity specified in the same *specification-part*, the type parameter or array bound  
34 shall be specified in a prior specification of the *specification-part*. The prior specification may be to the  
35 left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*.

#### NOTE 7.11

The following are examples of initialization expressions:

```
3
-3 + 4
'AB'
'AB' // 'CD'
```

**NOTE 7.11 (cont.)**

```

('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.

## 1 7.1.8 Evaluation of operations

2 An intrinsic operation requires the values of its operands.

3 The execution of any numeric operation whose result is not defined by the arithmetic used by the  
4 processor is prohibited. Raising a negative-valued primary of type real to a real power is prohibited.

5 The evaluation of a function reference shall neither affect nor be affected by the evaluation of any other  
6 entity within the statement. If a function reference causes definition or undefinition of an actual argument  
7 of the function, that argument or any associated entities shall not appear elsewhere in the same statement.  
8 However, execution of a function reference in the logical expression in an IF statement (8.1.3.4), the mask  
9 expression in a WHERE statement (7.4.3.1), or the subscripts and strides in a FORALL statement (7.4.4)  
10 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.

11 The declared type of an expression in which a function reference appears does not affect, and is not  
12 affected by, the evaluation of the actual arguments of the function.

13 Execution of an array element reference requires the evaluation of its subscripts. The type of an expres-  
14 sion in which the array element reference appears does not affect, and is not affected by, the evaluation  
15 of its subscripts. Execution of an array section reference requires the evaluation of its section subscripts.  
16 The type of an expression in which an array section appears does not affect, and is not affected by, the  
17 evaluation of the array section subscripts. Execution of a substring reference requires the evaluation of  
18 its substring expressions. The type of an expression in which a substring appears does not affect, and  
19 is not affected by, the evaluation of the substring expressions. It is not necessary for a processor to  
20 evaluate any subscript expressions or substring expressions for an array of zero size or a character entity  
21 of zero length.

22 The appearance of an array constructor requires the evaluation of each *scalar-int-expr* of the *ac-implied-*  
23 *do-control* in any *ac-implied-do* it may contain. The type of an expression in which an array constructor

- 1 appears does not affect, and is not affected by, the evaluation of such bounds and stride expressions.
- 2 When an elemental binary operation is applied to a scalar and an array or to two arrays of the same
- 3 shape, the operation is performed element-by-element on corresponding array elements of the array
- 4 operands.

**NOTE 7.13**

For example, the array expression

$$A + B$$

produces an array of the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, etc.

- 5 When an elemental unary operator operates on an array operand, the operation is performed element-
- 6 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.

### 7 7.1.8.1 Evaluation of operands

- 8 It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely
- 9 each operand, if the value of the expression can be determined otherwise.

**NOTE 7.15**

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.

- 10 If a statement contains a function reference in a part of an expression that need not be evaluated, all
- 11 entities that would have become defined in the execution of that reference become undefined at the
- 12 completion of evaluation of the expression containing the function reference.

**NOTE 7.16**

In the examples in Note 7.15, 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.

- 13 If a statement contains a function reference in a part of an expression that need not be evaluated, no
- 14 invocation of that function in that part of the expression shall execute an image control statement other

1 than CRITICAL or END CRITICAL.

#### NOTE 7.17

This restriction is intended to avoid inadvertant deadlock caused by optimization.

### 2 7.1.8.2 Integrity of parentheses

3 Subclauses 7.1.8.2 to 7.1.8.7 state certain conditions under which a processor may evaluate an expression  
4 that is different from the one specified by applying the rules given in 7.1.1 and 7.2. However, any  
5 expression in parentheses shall be treated as a data entity.

#### NOTE 7.18

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 7.1.8.3 Evaluation of numeric intrinsic operations

7 The rules given in 7.2.2 specify the interpretation of a numeric intrinsic operation. Once the interpreta-  
8 tion has been established in accordance with those rules, the processor may evaluate any mathematically  
9 equivalent expression, provided that the integrity of parentheses is not violated.

10 Two expressions of a numeric type are mathematically equivalent if, for all possible values of their  
11 primaries, their mathematical values are equal. However, mathematically equivalent expressions of  
12 numeric type may produce different computational results.

#### NOTE 7.19

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 7.2.2.1.

#### NOTE 7.20

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.

**NOTE 7.20 (cont.)**

<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 In addition to the parentheses required to establish the desired interpretation, parentheses may be  
2 included to restrict the alternative forms that may be used by the processor in the actual evaluation  
3 of the expression. This is useful for controlling the magnitude and accuracy of intermediate values  
4 developed during the evaluation of an expression.

**NOTE 7.21**

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.

- 5 Each operand in a numeric intrinsic operation has a type that may depend on the order of evaluation  
6 used by the processor.

**NOTE 7.22**

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.

#### 7 7.1.8.4 Evaluation of the character intrinsic operation

- 8 The rules given in 7.2.3 specify the interpretation of the character intrinsic operation. A processor is  
9 only required to evaluate as much of the character intrinsic operation as is required by the context in  
10 which the expression appears.

**NOTE 7.23**

For example, the statements

```
CHARACTER (LEN = 2) C1, C2, C3, CF
C1 = C2 // CF (C3)
```

**NOTE 7.23 (cont.)**

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.

**1 7.1.8.5 Evaluation of relational intrinsic operations**

2 The rules given in 7.2.4 specify the interpretation of relational intrinsic operations. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate  
3 any other expression that is relationally equivalent, provided that the integrity of parentheses in any  
4 expression is not violated.  
5

**NOTE 7.24**

For example, the processor may choose to evaluate the expression

$$I > J$$

where I and J are integer variables, as

$$J - I < 0$$

6 Two relational intrinsic operations are relationally equivalent if their logical values are equal for all  
7 possible values of their primaries.

**8 7.1.8.6 Evaluation of logical intrinsic operations**

9 The rules given in 7.2.5 specify the interpretation of logical intrinsic operations. Once the interpretation  
10 of an expression has been established in accordance with those rules, 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 7.25**

For example, for the variables L1, L2, and L3 of type logical, the processor may choose to evaluate the expression

$$L1 \text{ .AND. } L2 \text{ .AND. } L3$$

as

$$L1 \text{ .AND. } (L2 \text{ .AND. } L3)$$

13 Two expressions of type logical are logically equivalent if their values are equal for all possible values of  
14 their primaries.

**15 7.1.8.7 Evaluation of bits intrinsic operations**

16 The rules given in 7.2.6 specify the interpretation of bits intrinsic operations. Once the interpretation  
17 of an expression has been established in accordance with those rules, the processor may evaluate any  
18 other expression that is computationally equivalent, provided that the integrity of parentheses in any  
19 expression is not violated.

**NOTE 7.26**

For example, for the variables B1, B2, and B3 of type bits, the processor may choose to evaluate the expression

$$B1 \text{ .XOR. } B2 \text{ .XOR. } B3$$

as

$$B1 \text{ .XOR. } (B2 \text{ .XOR. } B3)$$

- 1 Two expressions of type bits are computationally equivalent if their values are equal for all possible  
2 values of their primaries.

### 3 **7.1.8.8 Evaluation of a defined operation**

- 4 The rules given in 7.2 specify the interpretation of a defined operation. Once the interpretation of an  
5 expression has been established in accordance with those rules, the processor may evaluate any other  
6 expression that is equivalent, provided that the integrity of parentheses is not violated.

- 7 Two expressions of derived type are equivalent if their values are equal for all possible values of their  
8 primaries.

## 9 **7.2 Interpretation of operations**

### 10 **7.2.1 General**

- 11 The intrinsic operations are those defined in 7.1.2. These operations are divided into the following  
12 categories: numeric, character, relational, logical, and bits. The interpretations defined in subclause 7.2  
13 apply to both scalars and arrays; the interpretation for arrays is obtained by applying the interpretation  
14 for scalars element by element.

- 15 The interpretation of a defined operation is provided by the function that defines the operation. The type,  
16 type parameters and interpretation of an expression that consists of an intrinsic or defined operation are  
17 independent of the type and type parameters of the context or any larger expression in which it appears.

**NOTE 7.27**

For example, if X is of type real, J is of type integer, and INT is the real-to-integer intrinsic conversion function, the expression INT (X + J) is an integer expression and X + J is a real expression.

- 18 The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT.,  
19 .LE., .GT., .GE., .EQ., and .NE., respectively.

### 20 **7.2.2 Numeric intrinsic operations**

- 21 A numeric operation is used to express a numeric computation. Evaluation of a numeric operation  
22 produces a numeric value. The permitted data types for operands of the numeric intrinsic operations  
23 are specified in 7.1.2.

- 24 The numeric operators and their interpretation in an expression are given in Table 7.2, where  $x_1$  denotes  
25 the operand to the left of the operator and  $x_2$  denotes the operand to the right of the operator.



Table 7.2: 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 $x_2$

- 1 The interpretation of a division operation depends on the types of the operands (7.2.2.1).
- 2 If  $x_1$  and  $x_2$  are of type integer and  $x_2$  has a negative value, the interpretation of  $x_1 ** x_2$  is the same
- 3 as the interpretation of  $1/(x_1 ** \text{ABS}(x_2))$ , which is subject to the rules of integer division (7.2.2.1).

**NOTE 7.28**

For example,  $2 ** (-3)$  has the value of  $1/(2 ** 3)$ , which is zero.

**4 7.2.2.1 Integer division**

- 5 One operand of type integer may be divided by another operand of type integer. Although the math-
- 6 ematical quotient of two integers is not necessarily an integer, Table 7.1 specifies that an expression
- 7 involving the division operator with two operands of type integer is interpreted as an expression of type
- 8 integer. The result of such an operation is the integer closest to the mathematical quotient and between
- 9 zero and the mathematical quotient inclusively.

**NOTE 7.29**

For example, the expression  $(-8) / 3$  has the value  $(-2)$ .

**10 7.2.2.2 Complex exponentiation**

- 11 In the case of a complex value raised to a complex power, the value of the operation  $x_1 ** x_2$  is the
- 12 principal value of  $x_1^{x_2}$ .

**13 7.2.3 Character intrinsic operation**

- 14 The character intrinsic operator `//` is used to concatenate two operands of type character with the same
- 15 kind type parameter. Evaluation of the character intrinsic operation produces a result of type character.
- 16 The interpretation of the character intrinsic operator `//` when used to form an expression is given in
- 17 Table 7.3, where  $x_1$  denotes the operand to the left of the operator and  $x_2$  denotes the operand to the
- 18 right of the operator.

Table 7.3: Interpretation of the character intrinsic operator `//`

Operator	Representing	Use of operator	Interpretation
//	Concatenation	$x_1 // x_2$	Concatenate $x_1$ with $x_2$

- 19 The result of the character intrinsic operation `//` is a character string whose value is the value of  $x_1$
- 20 concatenated on the right with the value of  $x_2$  and whose length is the sum of the lengths of  $x_1$  and  $x_2$ .
- 21 Parentheses used to specify the order of evaluation have no effect on the value of a character expression.

**NOTE 7.30**

For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

## 1 7.2.4 Relational intrinsic operations

- 2 A relational intrinsic operation is used to compare values of two operands using the relational intrinsic  
 3 operators .LT., .LE., .GT., .GE., .EQ., .NE., <, <=, >, >=, ==, and /=. The permitted types for  
 4 operands of the relational intrinsic operators are specified in 7.1.2.

**NOTE 7.31**

As shown in Table 7.1, 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.

- 5 Evaluation of a relational intrinsic operation produces a result of type default logical.  
 6 The interpretation of the relational intrinsic operators is given in Table 7.4, where  $x_1$  denotes the operand  
 7 to the left of the operator and  $x_2$  denotes the operand to the right of the operator.

Table 7.4: **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 $x_2$

- 8 A numeric relational intrinsic operation is interpreted as having the logical value true if and only if the  
 9 values of the operands satisfy the relation specified by the operator.

10 In the numeric relational operation

11 
$$x_1 \text{ rel-op } x_2$$

- 12 if the types or kind type parameters of  $x_1$  and  $x_2$  differ, their values are converted to the type and kind  
 13 type parameter of the expression  $x_1 + x_2$  before evaluation.

- 14 A character relational intrinsic operation is interpreted as having the logical value true if and only if the  
 15 values of the operands satisfy the relation specified by the operator.

- 16 For a character relational intrinsic operation, the operands are compared one character at a time in  
 17 order, beginning with the first character of each character operand. If the operands are of unequal  
 18 length, the shorter operand is treated as if it were extended on the right with blanks to the length of

1 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  
 2 the same as the character in the corresponding position in  $x_2$ ,  $x_1$  is equal to  $x_2$ . Otherwise, at the first  
 3 position where the character operands differ, the character operand  $x_1$  is considered to be less than  $x_2$   
 4 if the character value of  $x_1$  at this position precedes the value of  $x_2$  in the collating sequence (4.4.5.4);  
 5  $x_1$  is greater than  $x_2$  if the character value of  $x_1$  at this position follows the value of  $x_2$  in the collating  
 6 sequence.

#### NOTE 7.32

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.

7 A bits relational intrinsic operation is interpreted as having the logical value true if and only if the values  
 8 of the operands satisfy the relation specified by the operator.

9 For a bits relational intrinsic operation,  $x_1$  and  $x_2$  are equal if and only if each corresponding bit has  
 10 the same value. If  $x_1$  and  $x_2$  are not equal, and the leftmost unequal corresponding bit of  $x_1$  is 1 and  
 11  $x_2$  is 0 then  $x_1$  is greater than  $x_2$ ; otherwise  $x_1$  is less than  $x_2$ .

## 7.2.5 Logical intrinsic operations

13 A logical operation is used to express a logical computation. Evaluation of a logical operation produces  
 14 a result of type logical. The permitted types for operands of the logical intrinsic operations are specified  
 15 in 7.1.2.

16 The logical operators and their interpretation when used to form an expression are given in Table 7.5,  
 17 where  $x_1$  denotes the operand to the left of the operator and  $x_2$  denotes the operand to the right of the  
 18 operator.

Table 7.5: Interpretation of the logical intrinsic operators

Operator	Representing	Use of operator	Interpretation
.NOT.	Logical negation	.NOT. $x_2$	True if $x_2$ is false
.AND.	Logical conjunction	$x_1$ .AND. $x_2$	True if $x_1$ and $x_2$ are both true
.OR.	Logical inclusive disjunction	$x_1$ .OR. $x_2$	True if $x_1$ and/or $x_2$ is true
.EQV.	Logical equivalence	$x_1$ .EQV. $x_2$	True if both $x_1$ and $x_2$ are true or both are false
.NEQV.	Logical nonequivalence	$x_1$ .NEQV. $x_2$	True if either $x_1$ or $x_2$ is true, but not both
.XOR.	Logical nonequivalence	$x_1$ .XOR. $x_2$	True if either $x_1$ or $x_2$ is true, but not both

19 The values of the logical intrinsic operations are shown in Table 7.6.

Table 7.6: The values of operations involving logical intrinsic operators

$x_1$	$x_2$	.NOT. $x_2$	$x_1$ .AND. $x_2$	$x_1$ .OR. $x_2$	$x_1$ .EQV. $x_2$	$x_1$ .NEQV. $x_2$	$x_1$ .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

## 7.2.6 Bits intrinsic operations

Bit operations are used to express bitwise operations on sequences of bits, or to concatenate such sequences. Evaluation of a bits operation produces a result of type bits. The permitted types of operands of the bits intrinsic operations are specified in 7.1.2.

The bits operators and their interpretation when used to form an expression are given in Table 7.7, where  $x_1$  denotes the operand of type bits to the left of the operator and  $x_2$  denotes the operand of type bits to the right of the operator.

Table 7.7: Interpretation of the bits intrinsic operators

Operator	Representing	Use of operator	Interpretation
(//)	Concatenation	$x_1 // x_2$	Concatenation of $x_1$ and $x_2$
.NOT.	Bitwise NOT	.NOT. $x_2$	Bitwise NOT of $x_2$
.AND.	Bitwise AND	$x_1$ .AND. $x_2$	Bitwise AND of $x_1$ and $x_2$
.OR.	Bitwise inclusive OR	$x_1$ .OR. $x_2$	Bitwise OR of $x_1$ and $x_2$
.EQV.	Bitwise equivalence	$x_1$ .EQV. $x_2$	Bitwise equivalence of $x_1$ and $x_2$
.NEQV.	Bitwise nonequivalence	$x_1$ .NEQV. $x_2$	Bitwise nonequivalence of $x_1$ and $x_2$
.XOR.	Bitwise exclusive OR	$x_1$ .XOR. $x_2$	Bitwise exclusive OR of $x_1$ and $x_2$

The leftmost  $\text{KIND}(x_1)$  bits of the result of the bits concatenation operation are the value of  $x_1$  and the rightmost  $\text{KIND}(x_2)$  bits of the result are the value of  $x_2$ .

For a bits intrinsic operation other than //, the result value is computed separately for each pair of bits at corresponding positions in each operand. The value of each bit operation, for bits denoted  $b_1$  and  $b_2$  are given in Table 7.8.

Table 7.8: The values of bitwise operations involving bits 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. $x_2$
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

## 7.3 Precedence of operators

There is a precedence among the intrinsic and extension operations corresponding to the form of expressions specified in 7.1.1, which determines the order in which the operands are combined unless the order is changed by the use of parentheses. This precedence order is summarized in Table 7.9.

Table 7.9: 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.	.

Categories of operations and relative precedence (cont.)

Category of operation	Operators	Precedence
Logical, Bits	.AND.	.
Logical, Bits	.OR.	.
Logical, Bits	.EQV., .NEQV., .XOR.	.
Extension	<i>defined-binary-op</i>	Lowest

- 1 The precedence of a defined operation is that of its operator.

**NOTE 7.33**

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)`

- 2 The general form of an expression (7.1.1) also establishes a precedence among operators in the same
- 3 syntactic class. This precedence determines the order in which the operands are to be combined in
- 4 determining the interpretation of the expression unless the order is changed by the use of parentheses.

**NOTE 7.34**

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 (7.1.1), 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

**NOTE 7.34 (cont.)**

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 7.35**

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

```
L .OR. ((A + B) >= C)
```

**NOTE 7.36**

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 7.4 Assignment

2 Execution of an assignment statement causes a variable to become defined or redefined. Execution of a  
 3 pointer assignment statement causes a pointer to become associated with a target or causes its pointer  
 4 association status to become disassociated or undefined. Execution of a WHERE statement or WHERE  
 5 construct masks the evaluation of expressions and assignment of values in array assignment statements  
 6 according to the value of a logical array expression. Execution of a FORALL statement or FORALL  
 7 construct controls the assignment to elements of arrays by using a set of index variables and a mask  
 8 expression.

### 9 7.4.1 Assignment statement

10 A variable may be defined or redefined by execution of an assignment statement.

1 **7.4.1.1 General form**2 R734 *assignment-stmt* **is** *variable* = *expr*3 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)

```

4 An *assignment-stmt* shall meet the requirements of either a defined assignment statement or an intrinsic  
5 assignment statement.

6 **7.4.1.2 Intrinsic assignment statement**

7 An **intrinsic assignment statement** is an assignment statement that is not a defined assignment  
8 statement (7.4.1.4). In an intrinsic assignment statement,

- 9 (1) if the variable is polymorphic it shall be allocatable,  
10 (2) if *variable* is a co-indexed object, it shall not be of a type that has an allocatable ultimate  
11 component,

**J3 internal note****Unresolved Technical Issue 23**

06-174r3 had no edit about this at all; when it came up in discussion with JKR et al they suggested making a runtime requirement that the types and shapes of the allocatable components must match; however, that would be very unsafe. Since the only safe way of programming such an assignment would be to avoid the intrinsic assignment altogether, we should not allow the intrinsic assignment in this case. Furthermore, giving different semantics to intrinsic assignment when a component is a co-indexed object is a bad idea - it will confuse the users - better to disallow the discrepant inconsistency.

The fact that this inherently unsafe idea affects cross-image variables with all the fun of race conditions and other goodies to make debugging impossible simply underlines that this is a bad idea."

- 12 (3) if *expr* is an array then the variable shall also be an array,  
13 (4) the shapes of the variable and *expr* shall conform unless the variable is an allocatable array  
14 that has the same rank as *expr* and is neither a co-array nor a co-indexed object,  
15 (5) if the variable is an allocatable co-array or co-indexed object, its dynamic type shall be the  
16 same as that of *expr*,

**J3 internal note****Unresolved Technical Issue 24**

Since this almost completely guts the polymorphic assignment feature for co-things, why not just disallow polymorphic assignment altogether for co-things? That would be *MUCH SAFER*, and result in the loss of almost no functionality (one can point a non-polymorphic pointer at the polymorphic co-thing if one wishes to do the required assignment. Furthermore, doing that would still allow co-arrays to be extended later on to "do the right thing" for polymorphic assignment.

- 17 (6) if the variable is polymorphic it shall be type compatible with *expr* and have the same rank;  
18 otherwise the declared types of the variable and *expr* shall conform as specified in Table

- 1 7.10, and
- 2 (7) if the variable is of derived type each kind type parameter of the variable shall have the
- 3 same value as the corresponding type parameter of *expr*, and
- 4 (8) if the variable is of derived type each length type parameter of the variable shall have the
- 5 same value as the corresponding type parameter of *expr* unless the variable is allocatable
- 6 and its corresponding type parameter is deferred.

7 Table 7.10: **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

8 A **numeric intrinsic assignment statement** is an intrinsic assignment statement for which the vari-  
 9 able and *expr* are of numeric type. A **character intrinsic assignment statement** is an intrinsic  
 10 assignment statement for which the variable and *expr* are of type character. A **logical intrinsic as-**  
 11 **signment statement** is an intrinsic assignment statement for which the variable and *expr* are of type  
 12 logical. A **bits intrinsic assignment statement** is an intrinsic assignment statement for which either  
 13 the variable or *expr* is of type bits. A **derived-type intrinsic assignment statement** is an intrinsic  
 14 assignment statement for which the variable and *expr* are of derived type.

15 An **array intrinsic assignment statement** is an intrinsic assignment statement for which the variable  
 16 is an array. The the variable shall not be a many-one array section (6.2.2.3.2).

17 If the variable is a pointer, it shall be associated with a definable target such that the type, type  
 18 parameters, and shape of the target and *expr* conform.

#### 19 7.4.1.3 Interpretation of intrinsic assignments

20 Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all  
 21 expressions within *variable* (7.1.8), the possible conversion of *expr* to the type and type parameters of  
 22 the variable (Table 7.11), and the definition of the variable with the resulting value. The execution of  
 23 the assignment shall have the same effect as if the evaluation of *expr* and the evaluation of all expressions  
 24 in *variable* occurred before any portion of the variable is defined by the assignment. The evaluation of  
 25 expressions within *variable* shall neither affect nor be affected by the evaluation of *expr*. No value is  
 26 assigned to the variable if it is of type character and zero length, or is an array of size zero.

27 If the variable is a pointer, the value of *expr* is assigned to the target of the variable.

28 If the variable is an allocated allocatable variable, it is deallocated if *expr* is an array of different shape,  
 29 any of the corresponding length type parameter values of the variable and *expr* differ, or the dynamic  
 30 type of the variable and *expr* differ. If the variable is or becomes an unallocated allocatable variable,  
 31 then it is allocated with each deferred type parameter equal to the corresponding type parameter of *expr*,  
 32 with the shape of *expr*, with each lower bound equal to the corresponding element of LBOUND(*expr*),  
 33 and with the same dynamic 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.

- 1 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 value following the assignment is 'AABCDF'.

- 2 If *expr* is a scalar and the variable is an array, the *expr* is treated as if it were an array of the same
- 3 shape as the variable with every element of the array equal to the scalar value of *expr*.
- 4 If the variable is an array, the assignment is performed element-by-element on corresponding array
- 5 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.

- 6 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)
```

- 1 For a numeric intrinsic assignment statement, the variable and *expr* may have different numeric types
- 2 or different kind type parameters, in which case the value of *expr* is converted to the type and kind type
- 3 parameter of the variable according to the rules of Table 7.11.

Table 7.11: **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.	

- 4 For a logical intrinsic assignment statement, the variable and *expr* may have different kind type param-
- 5 eters, in which case the value of *expr* is converted to the kind type parameter of the variable.
- 6 For a character intrinsic assignment statement, the variable and *expr* may have different character length
- 7 parameters in which case the conversion of *expr* to the length of the variable is as follows.
- 8 (1) If the length of the variable is less than that of *expr*, the value of *expr* is truncated from
- 9 the right until it is the same length as the variable.
- 10 (2) If the length of the variable is greater than that of *expr*, the value of *expr* is extended on
- 11 the right with blanks until it is the same length as the variable.
- 12 If the variable and *expr* have different kind type parameters, each character *c* in *expr* is converted to
- 13 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.

- 14 For a bits intrinsic assignment statement, the variable and *expr* may have different types or different
- 15 kind type parameters, in which case the value of *expr* is converted to the type and kind type parameter
- 16 of the variable according to the rules of Table 7.12.

Table 7.12: **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, 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 copies it to the part of the result which occurs first in memory address order and leaves the remaining bits of the result processor-dependent, whereas bits assignment copies the source to the rightmost bits and makes the remaining bits all zero.

1 A derived-type intrinsic assignment is performed as if each component of the variable were assigned  
 2 from the corresponding component of *expr* using pointer assignment (7.4.2) for each pointer component,  
 3 defined assignment for each nonpointer nonallocatable component of a type that has a type-bound  
 4 defined assignment consistent with the component, intrinsic assignment for each co-array component, and  
 5 intrinsic assignment for each other nonpointer nonallocatable component. For a non-co-array allocatable  
 6 component the following sequence of operations is applied.

- 7 (1) If the component of the variable is allocated, it is deallocated.
- 8 (2) If the component of the value of *expr* is allocated, the corresponding component of the  
 9 variable is allocated with the same dynamic type and type parameters as the component  
 10 of the value of *expr*. If it is an array, it is allocated with the same bounds. The value of  
 11 the component of the value of *expr* is then assigned to the corresponding component of the  
 12 variable using defined assignment if the declared type of the component has a type-bound  
 13 defined assignment consistent with the component, and intrinsic assignment for the dynamic  
 14 type of that component otherwise.

15 The processor may perform the component-by-component assignment in any order or by any means that  
 16 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.

#### 17 7.4.1.4 Defined assignment statement

18 A **defined assignment statement** is an assignment statement that is defined by a subroutine and a  
 19 generic interface (4.5.2, 12.4.2.1.2) that specifies ASSIGNMENT (=). A **defined elemental assign-**

- 1 **ment statement** is a defined assignment statement for which the subroutine is elemental (12.8).
- 2 A subroutine defines the defined assignment  $x_1 = x_2$  if
- 3 (1) the subroutine is specified with a SUBROUTINE (12.6.2.2) or ENTRY (12.6.2.5) statement
  - 4 that specifies two dummy arguments,  $d_1$  and  $d_2$ ,
  - 5 (2) either
  - 6 (a) a generic interface (12.4.2.1) provides the subroutine with a *generic-spec* of ASSIGN-
  - 7 MENT (=), or
  - 8 (b) there is a generic binding (4.5.2) in the declared type of  $x_1$  or  $x_2$  with a *generic-spec*
  - 9 of ASSIGNMENT (=) and there is a corresponding binding to the subroutine in the
  - 10 dynamic type of  $x_1$  or  $x_2$ , respectively,
  - 11 (3) the types of  $d_1$  and  $d_2$  are compatible with the dynamic types of  $x_1$  and  $x_2$ , respectively,
  - 12 (4) the type parameters, if any, of  $d_1$  and  $d_2$  match the corresponding type parameters of  $x_1$
  - 13 and  $x_2$ , respectively, and
  - 14 (5) either
  - 15 (a) the ranks of  $x_1$  and  $x_2$  match those of  $d_1$  and  $d_2$  or
  - 16 (b) the subroutine is elemental,  $x_1$  and  $x_2$  are conformable, and there is no other subrou-
  - 17 tine that defines the operation.
- 18 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.

#### 19 7.4.1.5 Interpretation of defined assignment statements

20 The interpretation of a defined assignment is provided by the subroutine that defines it.

21 If the defined assignment is an elemental assignment and the the variable in the assignment is an array,  
 22 the defined assignment is performed element-by-element, on corresponding elements of the variable and  
 23 *expr*. If *expr* is a scalar, it is treated as if it were an array of the same shape as the variable with every  
 24 element of the array equal to the scalar value of *expr*.

#### NOTE 7.46

The rules of defined assignment (12.4.2.1.2), procedure references (12.5), subroutine references (12.5.3), and elemental subroutine arguments (12.8.3) ensure that the defined assignment has the same effect as if the evaluation of all operations in  $x_2$  and  $x_1$  occurs before any portion of  $x_1$  is defined. If an elemental assignment is defined by a pure elemental subroutine, the element assignments may be performed simultaneously or in any order.

#### 25 7.4.2 Pointer assignment

26 Pointer assignment causes a pointer to become associated with a target or causes its pointer association  
 27 status to become disassociated or undefined. Any previous association between the pointer and a target  
 28 is broken.

29 Pointer assignment for a pointer component of a structure may also take place by execution of a derived-  
 30 type intrinsic assignment statement (7.4.1.3).

31 A pointer may also become associated with a target by allocation of the pointer.

32 R735 *pointer-assignment-stmt*      **is** *data-pointer-object* [ (*bounds-spec-list*) ] => *data-target*  
 33                                            **or** *data-pointer-object* (*bounds-remapping-list*) => *data-target*  
 34                                            **or** *proc-pointer-object* => *proc-target*

- 1 R736 *data-pointer-object* **is** *variable-name*  
2 **or** *scalar-variable* % *data-pointer-component-name*
- 3 C717 (R735) If *data-target* is not unlimited polymorphic, either *data-pointer-object* shall be type  
4 compatible (4.3.1.3) with it and the corresponding kind type parameters shall be equal, or  
5 *data-target* and *data-pointer-object* shall be bits compatible.
- 6 C718 (R735) If *data-target* is unlimited polymorphic, *data-pointer-object* shall be unlimited polymor-  
7 phic, of a sequence derived type, or of a type with the BIND attribute.
- 8 C719 (R735) If *bounds-spec-list* is specified, the number of *bounds-specs* shall equal the rank of *data-*  
9 *pointer-object*.
- 10 C720 (R735) If *bounds-remapping-list* is specified, the number of *bounds-remappings* shall equal the  
11 rank of *data-pointer-object*.
- 12 C721 (R735) If *bounds-remapping-list* is not specified, the ranks of *data-pointer-object* and *data-target*  
13 shall be the same.
- 14 C722 (R736) A *variable-name* shall have the POINTER attribute.
- 15 C723 (R736) A *scalar-variable* shall be a *data-ref*.
- 16 C724 (R736) A *data-pointer-component-name* shall be the name of a component of *scalar-variable*  
17 that is a data pointer.
- 18 C725 (R736) A *data-pointer-object* shall not be a co-indexed object.
- 19 R737 *bounds-spec* **is** *lower-bound-expr* :  
20 R738 *bounds-remapping* **is** *lower-bound-expr* : *upper-bound-expr*  
21 R739 *data-target* **is** *variable*  
22 **or** *expr*
- 23 C726 (R739) A *variable* shall have either the TARGET or POINTER attribute, and shall not be an  
24 array section with a vector subscript.
- 25 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.

- 26 C728 (R739) An *expr* shall be a reference to a function whose result is a data pointer.
- 27 R740 *proc-pointer-object* **is** *proc-pointer-name*  
28 **or** *proc-component-ref*
- 29 R741 *proc-component-ref* **is** *scalar-variable* % *procedure-component-name*
- 30 C729 (R741) The *scalar-variable* shall be a *data-ref*.
- 31 C730 (R741) The *procedure-component-name* shall be the name of a procedure pointer component of  
32 the declared type of *scalar-variable*.
- 33 R742 *proc-target* **is** *expr*

- 1                                    **or** *procedure-name*  
 2                                    **or** *proc-component-ref*

3 C731 (R742) An *expr* shall be a reference to a function whose result is a procedure pointer.

4 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 (●).

7 C733 (R742) The *proc-target* shall not be a nonintrinsic elemental procedure.

#### 8 7.4.2.1 Data pointer assignment

9 If *data-pointer-object* is not polymorphic and *data-target* is polymorphic with dynamic type that differs from its declared type, the assignment target is the ancestor component of *data-target* that has the type of *data-pointer-object*. Otherwise, the assignment target is *data-target*.

12 If *data-target* is not a pointer, *data-pointer-object* becomes pointer associated with the assignment target. Otherwise, the pointer association status of *data-pointer-object* becomes that of *data-target*; if *data-target* is associated with an object, *data-pointer-object* becomes associated with the assignment target. If *data-target* is allocatable, it shall be allocated.

16 If *data-pointer-object* is polymorphic (4.3.1.3), it assumes the dynamic type of *data-target*. If *data-pointer-object* is of sequence derived type or a type with the BIND attribute, the dynamic type of *data-target* shall be that derived type.

19 If *data-target* is a disassociated pointer, all nondeferred type parameters of the declared type of *data-pointer-object* that correspond to nondeferred type parameters of *data-target* shall have the same values as the corresponding type parameters of *data-target*, or *data-pointer-object* and *data-target* shall be bits compatible (4.3.1.3).

23 Otherwise, all nondeferred type parameters of the declared type of *data-pointer-object* shall have the same values as the corresponding type parameters of *data-target*.

25 If *data-pointer-object* has nondeferred type parameters that correspond to deferred type parameters of *data-target*, *data-target* shall not be a pointer with undefined association status.

27 If *data-pointer-object* has the CONTIGUOUS attribute, *data-target* shall be contiguous.

28 If *bounds-remapping-list* is specified, *data-target* shall be contiguous (5.3.6) or of rank one. It shall not be a disassociated or undefined pointer, and the size of *data-target* shall not be less than the size of *data-pointer-object*. The elements of the target of *data-pointer-object*, in array element order (6.2.2.2), are the first SIZE(*data-pointer-object*) elements of *data-target*.

32 If no *bounds-remapping-list* is specified, the extent of a dimension of *data-pointer-object* is the extent of the corresponding dimension of *data-target*. If *bounds-spec-list* appears, it specifies the lower bounds; otherwise, the lower bound of each dimension is the result of the intrinsic function LBOUND (13.7.97) applied to the corresponding dimension of *data-target*. The upper bound of each dimension is one less than the sum of the lower bound and the extent.

#### 37 7.4.2.2 Procedure pointer assignment

38 If the *proc-target* is not a pointer, *proc-pointer-object* becomes pointer associated with *proc-target*. Otherwise, the pointer association status of *proc-pointer-object* becomes that of *proc-target*; if *proc-target* is associated with a procedure, *proc-pointer-object* becomes associated with the same procedure.

41 If *proc-target* is the name of an internal procedure the **host instance** of *proc-pointer-object* becomes

- 1 the innermost currently executing instance of the host procedure. Otherwise if *proc-target* has a host
- 2 instance the host instance of *proc-pointer-object* becomes that instance. Otherwise *proc-pointer-object*
- 3 has no host instance.
  
- 4 If *proc-pointer-object* has an explicit interface, its characteristics shall be the same as *proc-target* except
- 5 that *proc-target* may be pure even if *proc-pointer-object* is not pure and *proc-target* may be an elemental
- 6 intrinsic procedure even if *proc-pointer-object* is not elemental.
  
- 7 If the characteristics of *proc-pointer-object* or *proc-target* are such that an explicit interface is required,
- 8 both *proc-pointer-object* and *proc-target* shall have an explicit interface.
  
- 9 If *proc-pointer-object* has an implicit interface and is explicitly typed or referenced as a function, *proc-*
- 10 *target* shall be a function. If *proc-pointer-object* has an implicit interface and is referenced as a subroutine,
- 11 *proc-target* shall be a subroutine.
  
- 12 If *proc-target* and *proc-pointer-object* are functions, they shall have the same type; corresponding type
- 13 parameters shall either both be deferred or both have the same value.
  
- 14 If *procedure-name* is a specific procedure name that is also a generic name, only the specific procedure
- 15 is associated with pointer-object.

### 16 7.4.2.3 Examples

#### NOTE 7.48

The following are examples of pointer assignment statements. (See Note 12.14 for declarations of P and BESSEL.)

```
NEW_NODE % LEFT => CURRENT_NODE
SIMPLE_NAME => TARGET_STRUCTURE % SUBSTRUCT % COMPONENT
PTR => NULL ( )
ROW => MAT2D (N, :)
WINDOW => MAT2D (I-1:I+1, J-1:J+1)
POINTER_OBJECT => POINTER_FUNCTION (ARG_1, ARG_2)
EVERY_OTHER => VECTOR (1:N:2)
WINDOW2 (0:, 0:) => MAT2D (ML:MU, NL:NU)
! P is a procedure pointer and BESSEL is a procedure with a
! compatible interface.
P => BESSEL

! Likewise for a structure component.
STRUCT % COMPONENT => BESSEL
```

#### NOTE 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
```

**NOTE 7.49 (cont.)**

```
VIEW_DIAG => MYDATA( 1::NR+1 )      ! The diagonal of MATRIX
```

Rows, columns, or blocks of the matrix can be accessed as sections of MATRIX.

**7.4.3 Masked array assignment – WHERE**

The masked array assignment is used to mask the evaluation of expressions and assignment of values in array assignment statements, according to the value of a logical array expression.

**7.4.3.1 General form of the masked array assignment**

A **masked array assignment** is either a WHERE statement or a WHERE construct.

R743	<i>where-stmt</i>	<b>is</b>	WHERE ( <i>mask-expr</i> ) <i>where-assignment-stmt</i>
R744	<i>where-construct</i>	<b>is</b>	<i>where-construct-stmt</i>
			[ <i>where-body-construct</i> ] ...
			[ <i>masked-elsewhere-stmt</i>
			[ <i>where-body-construct</i> ] ... ] ...
			[ <i>elsewhere-stmt</i>
			[ <i>where-body-construct</i> ] ... ]
			<i>end-where-stmt</i>
R745	<i>where-construct-stmt</i>	<b>is</b>	[ <i>where-construct-name</i> :] WHERE ( <i>mask-expr</i> )
R746	<i>where-body-construct</i>	<b>is</b>	<i>where-assignment-stmt</i>
		<b>or</b>	<i>where-stmt</i>
		<b>or</b>	<i>where-construct</i>
R747	<i>where-assignment-stmt</i>	<b>is</b>	<i>assignment-stmt</i>
R748	<i>mask-expr</i>	<b>is</b>	<i>logical-expr</i>
R749	<i>masked-elsewhere-stmt</i>	<b>is</b>	ELSEWHERE ( <i>mask-expr</i> ) [ <i>where-construct-name</i> ]
R750	<i>elsewhere-stmt</i>	<b>is</b>	ELSEWHERE [ <i>where-construct-name</i> ]
R751	<i>end-where-stmt</i>	<b>is</b>	END WHERE [ <i>where-construct-name</i> ]

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.

If a *where-construct* contains a *where-stmt*, a *masked-elsewhere-stmt*, or another *where-construct* then each *mask-expr* within the *where-construct* shall have the same shape. In each *where-assignment-stmt*, the *mask-expr* and the variable being defined shall be arrays of the same shape.

**NOTE 7.50**

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
```



**NOTE 7.50 (cont.)**

```

ELSEWHERE
    RAINING = .TRUE.
END WHERE

```

**1 7.4.3.2 Interpretation of masked array assignments**

2 When a WHERE statement or a *where-construct-stmt* is executed, a control mask is established. In  
3 addition, when a WHERE construct statement is executed, a pending control mask is established. If  
4 the statement does not appear as part of a *where-body-construct*, the *mask-expr* of the statement is  
5 evaluated, and the control mask is established to be the value of *mask-expr*. The pending control mask  
6 is established to have the value *.NOT. mask-expr* upon execution of a WHERE construct statement that  
7 does not appear as part of a *where-body-construct*. The *mask-expr* is evaluated only once.

8 Each statement in a WHERE construct is executed in sequence.

9 Upon execution of a *masked-elsewhere-stmt*, the following actions take place in sequence.

- 10 (1) The control mask  $m_c$  is established to have the value of the pending control mask.
- 11 (2) The pending control mask is established to have the value  $m_c$  *.AND. (.NOT. mask-expr)*.
- 12 (3) The control mask  $m_c$  is established to have the value  $m_c$  *.AND. mask-expr*.

13 The *mask-expr* is evaluated at most once.

14 Upon execution of an ELSEWHERE statement, the control mask is established to have the value of the  
15 pending control mask. No new pending control mask value is established.

16 Upon execution of an ENDWHERE statement, the control mask and pending control mask are es-  
17 tablished to have the values they had prior to the execution of the corresponding WHERE construct  
18 statement. Following the execution of a WHERE statement that appears as a *where-body-construct*, the  
19 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.

20 Upon execution of a WHERE construct statement that is part of a *where-body-construct*, the pending  
21 control mask is established to have the value  $m_c$  *.AND. (.NOT. mask-expr)*. The control mask is then  
22 established to have the value  $m_c$  *.AND. mask-expr*. The *mask-expr* is evaluated at most once.

- 1 Upon execution of a WHERE statement that is part of a *where-body-construct*, the control mask is  
 2 established to have the value  $m_c$  .AND. *mask-expr*. The pending mask is not altered.
- 3 If a nonelemental function reference occurs in the *expr* or *variable* of a *where-assignment-stmt* or in a  
 4 *mask-expr*, the function is evaluated without any masked control; that is, all of its argument expressions  
 5 are fully evaluated and the function is fully evaluated. If the result is an array and the reference is not  
 6 within the argument list of a nonelemental function, elements corresponding to true values in the control  
 7 mask are selected for use in evaluating the *expr*, *variable* or *mask-expr*.
- 8 If an elemental operation or function reference occurs in the *expr* or *variable* of a *where-assignment-stmt*  
 9 or in a *mask-expr*, and is not within the argument list of a nonelemental function reference, the operation  
 10 is performed or the function is evaluated only for the elements corresponding to true values of the control  
 11 mask.
- 12 If an array constructor appears in a *where-assignment-stmt* or in a *mask-expr*, the array constructor is  
 13 evaluated without any masked control and then the *where-assignment-stmt* is executed or the *mask-expr*  
 14 is evaluated.
- 15 When a *where-assignment-stmt* is executed, the values of *expr* that correspond to true values of the  
 16 control mask are assigned to the corresponding elements of the variable.
- 17 The value of the control mask is established by the execution of a WHERE statement, a WHERE con-  
 18 struct statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE  
 19 statement. Subsequent changes to the value of entities in a *mask-expr* have no effect on the value of the  
 20 control mask. The execution of a function reference in the mask expression of a WHERE statement is  
 21 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

```

**7.4.4 FORALL**

- 23 FORALL constructs and statements are used to control the execution of assignment and pointer assign-  
 24 ment statements with selection by sets of index values and an optional mask expression.

**7.4.4.1 The FORALL Construct**

- 26 The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested  
 27 FORALL constructs and statements to be controlled by a single *forall-triplet-spec-list* and *scalar-mask-*  
 28 *expr*.

- |         |                              |           |                                                                                                |
|---------|------------------------------|-----------|------------------------------------------------------------------------------------------------|
| 29 R752 | <i>forall-construct</i>      | <b>is</b> | <i>forall-construct-stmt</i><br>[ <i>forall-body-construct</i> ] ...<br><i>end-forall-stmt</i> |
| 32 R753 | <i>forall-construct-stmt</i> | <b>is</b> | [ <i>forall-construct-name</i> :] FORALL <i>forall-header</i>                                  |
| 33 R754 | <i>forall-header</i>         | <b>is</b> | ( <i>forall-triplet-spec-list</i> [, <i>scalar-mask-expr</i> ] )                               |
| 34 R755 | <i>forall-triplet-spec</i>   | <b>is</b> | <i>index-name</i> = <i>subscript</i> : <i>subscript</i> [ : <i>stride</i> ]                    |
| 35 R619 | <i>subscript</i>             | <b>is</b> | <i>scalar-int-expr</i>                                                                         |
| 36 R622 | <i>stride</i>                | <b>is</b> | <i>scalar-int-expr</i>                                                                         |

- 1 R756 *forall-body-construct* **is** *forall-assignment-stmt*  
 2 **or** *where-stmt*  
 3 **or** *where-construct*  
 4 **or** *forall-construct*  
 5 **or** *forall-stmt*  
 6 R757 *forall-assignment-stmt* **is** *assignment-stmt*  
 7 **or** *pointer-assignment-stmt*  
 8 R758 *end-forall-stmt* **is** END FORALL [*forall-construct-name* ]
- 9 C737 (R758) If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have  
 10 the same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-*  
 11 *construct-stmt* shall have the same *forall-construct-name*.
- 12 C738 (R754) The *scalar-mask-expr* shall be scalar and of type logical.
- 13 C739 (R754) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined  
 14 operation, shall be a pure procedure (12.7).
- 15 C740 (R755) The *index-name* shall be a named scalar variable of type integer.
- 16 C741 (R755) A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-*  
 17 *name* in the *forall-triplet-spec-list* in which it appears.
- 18 C742 (R756) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-*  
 19 *construct*.
- 20 C743 (R756) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined  
 21 operation, assignment, or finalization, shall be a pure procedure.
- 22 C744 (R756) A *forall-body-construct* shall not be a branch target.
- 23 C745 (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)
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 7.4.4.2 Execution of the FORALL construct

- 5 There are three stages in the execution of a FORALL construct:

- 6 (1) determination of the values for *index-name* variables,
- 7 (2) evaluation of the *scalar-mask-expr*, and
- 8 (3) execution of the FORALL body constructs.

#### 1 7.4.4.2.1 Determination of the values for index variables

2 The subscript and stride expressions in the *forall-triplet-spec-list* are evaluated. These expressions may  
3 be evaluated in any order. The set of values that a particular *index-name* variable assumes is determined  
4 as follows.

- 5 (1) The lower bound  $m_1$ , the upper bound  $m_2$ , and the stride  $m_3$  are of type integer with the  
6 same kind type parameter as the *index-name*. Their values are established by evaluating  
7 the first subscript, the second subscript, and the stride expressions, respectively, including,  
8 if necessary, conversion to the kind type parameter of the *index-name* according to the rules  
9 for numeric conversion (Table 7.11). If a stride does not appear,  $m_3$  has the value 1. The  
10 value  $m_3$  shall not be zero.
- 11 (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  
12 of the construct is complete. Otherwise, the set of values for the *index-name* is  
13 
$$m_1 + (k - 1) \times m_3 \quad \text{where } k = 1, 2, \dots, max.$$

14 The set of combinations of *index-name* values is the Cartesian product of the sets defined by each triplet  
15 specification. An *index-name* becomes defined when this set is evaluated.

#### NOTE 7.57

The *stride* may be positive or negative; the FORALL body constructs are executed as long as  $max > 0$ . For the *forall-triplet-spec*

```
I = 10:1:-1
```

*max* has the value 10

#### 16 7.4.4.2.2 Evaluation of the mask expression

17 The *scalar-mask-expr*, if any, is evaluated for each combination of *index-name* values. If the *scalar-*  
18 *mask-expr* is not present, it is as if it were present with the value true. The *index-name* variables may  
19 be primaries in the *scalar-mask-expr*.

20 The **active combination of *index-name* values** is defined to be the subset of all possible combinations  
21 (7.4.4.2.1) for which the *scalar-mask-expr* has the value true.

#### NOTE 7.58

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)
. . .
```

#### 22 7.4.4.2.3 Execution of the FORALL body constructs

23 The *forall-body-constructs* are executed in the order in which they appear. Each construct is executed  
24 for all active combinations of the *index-name* values with the following interpretation:

25 Execution of a *forall-assignment-stmt* that is an *assignment-stmt* causes the evaluation of *expr* and all  
26 expressions within *variable* for all active combinations of *index-name* values. These evaluations may be  
27 done in any order. After all these evaluations have been performed, each *expr* value is assigned to the  
28 corresponding *variable*. The assignments may occur in any order.

29 Execution of a *forall-assignment-stmt* that is a *pointer-assignment-stmt* causes the evaluation of all  
30 expressions within *data-target* and *data-pointer-object* or *proc-target* and *proc-pointer-object*, the de-  
31 termination of any pointers within *data-pointer-object* or *proc-pointer-object*, and the determination of

- 1 the target for all active combinations of *index-name* values. These evaluations may be done in any
- 2 order. After all these evaluations have been performed, each *data-pointer-object* or *proc-pointer-object*
- 3 is associated with the corresponding target. These associations may occur in any order.
- 4 In a *forall-assignment-stmt*, a defined assignment subroutine shall not reference any *variable* that be-
- 5 comes defined by the statement.

**NOTE 7.59**

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)
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
```

- 6 Each statement in a *where-construct* (7.4.3) within a *forall-construct* is executed in sequence. When
- 7 a *where-stmt*, *where-construct-stmt* or *masked-elsewhere-stmt* is executed, the statement's *mask-expr* is
- 8 evaluated for all active combinations of *index-name* values as determined by the outer *forall-constructs*,
- 9 masked by any control mask corresponding to outer *where-constructs*. Any *where-assignment-stmt* is
- 10 executed for all active combinations of *index-name* values, masked by the control mask in effect for the
- 11 *where-assignment-stmt*.

**NOTE 7.60**

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

```
A =
  0  0  0  0
  1  1  1  0
  2  2  0  2
  1  0  2  3
  0  0  0  0
```

the results will be

```
A =
  1  1  1  1
  1  1  1  2
  2  2  3  2
B =
  1  1  1  1
  2  2  2  1
  1  1  1  1
```

**NOTE 7.60 (cont.)**

1	4	2	3	4	1	2	1
5	5	5	5	1	1	1	1

For an example of a FORALL construct containing a WHERE construct with an ELSEWHERE statement, see C.4.5.

- 1 Execution of a *forall-stmt* or *forall-construct* causes the evaluation of the *subscript* and *stride* expressions
- 2 in the *forall-triplet-spec-list* for all active combinations of the *index-name* values of the outer FORALL
- 3 construct. The set of combinations of *index-name* values for the inner FORALL is the union of the sets
- 4 defined by these bounds and strides for each active combination of the outer *index-name* values; it also
- 5 includes the outer *index-name* values. The *scalar-mask-expr* is then evaluated for all combinations of the
- 6 *index-name* values of the inner construct to produce a set of active combinations for the inner construct.
- 7 If there is no *scalar-mask-expr*, it is as if it were present with the value .TRUE.. Each statement in the
- 8 inner FORALL is then executed for each active combination of the *index-name* values.

**NOTE 7.61**

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

```

**9 7.4.4.3 The FORALL statement**

- 10 The FORALL statement allows a single assignment statement or pointer assignment to be controlled by
- 11 a set of index values and an optional mask expression.
- 12 R759 *forall-stmt* **is** FORALL *forall-header forall-assignment-stmt*
- 13 A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct*
- 14 that is a *forall-assignment-stmt*.
- 15 The scope of an *index-name* in a *forall-stmt* is the statement itself (16.4).

**NOTE 7.62**

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) 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 7.61 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 7.4.4.4 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 7.63**

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.

19 Any of these constructs may be named. If a construct is named, the name shall be the first lexical token  
 20 of the first statement of the construct and the last lexical token of the construct. In fixed source form, the  
 21 name preceding the construct shall be placed after character position 6.

22 Except for a CYCLE statement, a statement **belongs** to the innermost construct in which it appears  
 23 unless it contains a construct name, in which case it belongs to the named construct.

#### 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
```

## 1 8.1.2 Rules governing blocks

### 2 8.1.2.1 Executable constructs in blocks

3 If a block contains an executable construct, the executable construct shall be entirely within the block.

### 4 8.1.2.2 Control flow in blocks

5 Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a  
6 block and transfers from the interior of a block to outside the block may occur.

7 Subroutine and function references (12.5.2, 12.5.3) may appear in a block.

### 8 8.1.2.3 Execution of a block

9 Execution of a block begins with the execution of the first executable construct in the block. Execution  
10 of the block is completed when the last executable construct in the sequence is executed or when a  
11 branch out of the block takes place.

#### 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.

## 12 8.1.3 IF construct

13 The **IF construct** selects for execution at most one of its constituent blocks. The selection is based  
14 on a sequence of logical expressions. The **IF statement** controls the execution of a single statement  
15 (8.1.3.4) based on a single logical expression.

### 16 8.1.3.1 Form of the IF construct

17	R802	<i>if-construct</i>	<b>is</b>	<i>if-then-stmt</i> <i>block</i> [ <i>else-if-stmt</i> <i>block</i> ] ... [ <i>else-stmt</i> <i>block</i> ] <i>end-if-stmt</i>
24	R803	<i>if-then-stmt</i>	<b>is</b>	[ <i>if-construct-name</i> : ] IF ( <i>scalar-logical-expr</i> ) THEN
25	R804	<i>else-if-stmt</i>	<b>is</b>	ELSE IF ( <i>scalar-logical-expr</i> ) THEN [ <i>if-construct-name</i> ]
26	R805	<i>else-stmt</i>	<b>is</b>	ELSE [ <i>if-construct-name</i> ]
27	R806	<i>end-if-stmt</i>	<b>is</b>	END IF [ <i>if-construct-name</i> ]

28 C801 (R802) If the *if-then-stmt* of an *if-construct* specifies an *if-construct-name*, the corresponding  
29 *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* does  
30 not specify an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-*  
31 *name*. If an *else-if-stmt* or *else-stmt* specifies an *if-construct-name*, the corresponding *if-then-*  
32 *stmt* shall specify the same *if-construct-name*.

### 33 8.1.3.2 Execution of an IF construct

34 At most one of the blocks in the IF construct is executed. If there is an ELSE statement in the construct,  
35 exactly one of the blocks in the construct is executed. The scalar logical expressions are evaluated in  
36 the order of their appearance in the construct until a true value is found or an ELSE statement or END  
37 IF statement is encountered. If a true value or an ELSE statement is found, the block immediately

1 following is executed and this completes the execution of the construct. The scalar logical expressions  
 2 in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated  
 3 expressions is true and there is no ELSE statement, the execution of the construct is completed without  
 4 the execution of any block within the construct.

5 An ELSE IF statement or an ELSE statement shall not be a branch target statement. It is permissible  
 6 to branch to an END IF statement only from within its IF construct. Execution of an END IF statement  
 7 has no effect.

### 8 8.1.3.3 Examples of IF constructs

#### NOTE 8.4

```

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

```

### 9 8.1.3.4 IF statement

10 The IF statement controls a single action statement (R214).

11 R807 *if-stmt* **is** IF ( *scalar-logical-expr* ) *action-stmt*

12 C802 (R807) The *action-stmt* in the *if-stmt* shall not be an *if-stmt*, *end-program-stmt*, *end-function-*  
 13 *stmt*, or *end-subroutine-stmt*.

14 Execution of an IF statement causes evaluation of the scalar logical expression. If the value of the  
 15 expression is true, the action statement is executed. If the value is false, the action statement is not  
 16 executed and execution continues.

17 The execution of a function reference in the scalar logical expression may affect entities in the action  
 18 statement.

#### NOTE 8.5

An example of an IF statement is:

## NOTE 8.5 (cont.)

IF (A > 0.0) A = LOG (A)
--------------------------

1 **8.1.4 CASE construct**

2 The **CASE construct** selects for execution at most one of its constituent blocks. The selection is based  
 3 on the value of an expression.

4 **8.1.4.1 Form of the CASE construct**

- 5 R808 *case-construct*                      **is** *select-case-stmt*  
 6                                                              [ *case-stmt*  
 7                                                                      *block* ] ...  
 8                                                              *end-select-stmt*
- 9 R809 *select-case-stmt*                      **is** [ *case-construct-name* : ] SELECT CASE ( *case-expr* )  
 10 R810 *case-stmt*                              **is** CASE *case-selector* [*case-construct-name*]  
 11 R811 *end-select-stmt*                      **is** END SELECT [ *case-construct-name* ]
- 12 C803 (R808) If the *select-case-stmt* of a *case-construct* specifies a *case-construct-name*, the corre-  
 13 sponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt*  
 14 of a *case-construct* does not specify a *case-construct-name*, the corresponding *end-select-stmt*  
 15 shall not specify a *case-construct-name*. If a *case-stmt* specifies a *case-construct-name*, the  
 16 corresponding *select-case-stmt* shall specify the same *case-construct-name*.
- 17 R812 *case-expr*                              **is** *scalar-int-expr*  
 18                                                              **or** *scalar-char-expr*  
 19                                                              **or** *scalar-logical-expr*
- 20 R813 *case-selector*                          **is** ( *case-value-range-list* )  
 21                                                              **or** DEFAULT
- 22 C804 (R808) No more than one of the selectors of one of the CASE statements shall be DEFAULT.
- 23 R814 *case-value-range*                      **is** *case-value*  
 24                                                              **or** *case-value* :  
 25                                                              **or** : *case-value*  
 26                                                              **or** *case-value* : *case-value*
- 27 R815 *case-value*                              **is** *scalar-int-initialization-expr*  
 28                                                              **or** *scalar-char-initialization-expr*  
 29                                                              **or** *scalar-logical-initialization-expr*
- 30 C805 (R808) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For  
 31 character type, the kind type parameters shall be the same; character length differences are  
 32 allowed.
- 33 C806 (R808) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.
- 34 C807 (R808) For a given *case-construct*, the *case-value-ranges* shall not overlap; that is, there shall  
 35 be no possible value of the *case-expr* that matches more than one *case-value-range*.

36 **8.1.4.2 Execution of a CASE construct**

37 The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting  
 38 value is called the **case index**. For a case value range list, a match occurs if the case index matches any  
 39 of the case value ranges in the list. For a case index with a value of *c*, a match is determined as follows.

- 1       (1) If the case value range contains a single value *v* without a colon, a match occurs for type  
 2       logical if the expression *c* .EQV. *v* is true, and a match occurs for type integer or character  
 3       if the expression *c* == *v* is true.
- 4       (2) If the case value range is of the form *low* : *high*, a match occurs if the expression *low* <= *c*  
 5       .AND. *c* <= *high* is true.
- 6       (3) If the case value range is of the form *low* :, a match occurs if the expression *low* <= *c* is true.
- 7       (4) If the case value range is of the form : *high*, a match occurs if the expression *c* <= *high* is  
 8       true.
- 9       (5) If no other selector matches and a DEFAULT selector appears, it matches the case index.
- 10      (6) If no other selector matches and the DEFAULT selector does not appear, there is no match.
- 11 The block following the CASE statement containing the matching selector, if any, is executed. This  
 12 completes execution of the construct.
- 13 At most one of the blocks of a CASE construct is executed.
- 14 A CASE statement shall not be a branch target statement. It is permissible to branch to an *end-select-*  
 15 *stmt* only from within its CASE construct.

#### 16 8.1.4.3 Examples of CASE constructs

##### NOTE 8.6

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 8.7

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

```

**NOTE 8.7 (cont.)**

```

    END SELECT CHECK_PARENS
END DO SCAN_LINE
IF (LEVEL > 0) THEN
    PRINT *, 'MISSING RIGHT PARENTHESIS'
END IF

```

**NOTE 8.8**

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 8.9**

A code fragment showing several selections of one block:

```

SELECT CASE (N)
CASE (1, 3:5, 8) ! Selects 1, 3, 4, 5, 8
    CALL SUB
CASE DEFAULT
    CALL OTHER
END SELECT

```

**1 8.1.5 ASSOCIATE construct**

2 The ASSOCIATE construct associates named entities with expressions or variables during the execution  
3 of its block. These named construct entities (16.4) are associating entities (16.5.1.6). The names are  
4 **associate names**.

**5 8.1.5.1 Form of the ASSOCIATE construct**

6	R816	<i>associate-construct</i>	<b>is</b>	<i>associate-stmt</i> <i>block</i> <i>end-associate-stmt</i>
7				
8				
9	R817	<i>associate-stmt</i>	<b>is</b>	[ <i>associate-construct-name</i> : ] ASSOCIATE ■
10				■ ( <i>association-list</i> )
11	R818	<i>association</i>	<b>is</b>	<i>associate-name</i> => <i>selector</i>

1 R819 *selector* **is** *expr*  
 2 **or** *variable*

3 C808 (R818) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name*  
 4 shall not appear in a variable definition context (16.6.7).

5 C809 (R818) An *associate-name* shall not be the same as another *associate-name* in the same *associate-*  
 6 *stmt*.

7 R820 *end-associate-stmt* **is** END ASSOCIATE [ *associate-construct-name* ]

8 C810 (R820) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*,  
 9 the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the  
 10 *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the cor-  
 11 responding *end-associate-stmt* shall not specify an *associate-construct-name*.

#### 12 8.1.5.2 Execution of the ASSOCIATE construct

13 Execution of an ASSOCIATE construct causes execution of its *associate-stmt* followed by execution  
 14 of its block. During execution of that block each associate name identifies an entity, which is associ-  
 15 ated (16.5.1.6) with the corresponding selector. The associating entity assumes the declared type and  
 16 type parameters of the selector. If and only if the selector is polymorphic, the associating entity is  
 17 polymorphic.

18 The other attributes of the associating entity are described in 8.1.5.3.

19 It is permissible to branch to an *end-associate-stmt* only from within its ASSOCIATE construct.

#### 20 8.1.5.3 Attributes of associate names

21 Within a SELECT TYPE or ASSOCIATE construct, each associating entity has the same rank as its  
 22 associated selector. The lower bound of each dimension is the result of the intrinsic function LBOUND  
 23 (13.7.97) applied to the corresponding dimension of *selector*. The upper bound of each dimension is one  
 24 less than the sum of the lower bound and the extent. The associating entity has the ASYNCHRONOUS  
 25 or VOLATILE attribute if and only if the selector is a variable and has the attribute. The associating  
 26 entity has the TARGET attribute if and only if the selector is a variable and has either the TARGET  
 27 or POINTER attribute. If the associating entity is polymorphic, it assumes the dynamic type and type  
 28 parameter values of the selector. If the selector has the OPTIONAL attribute, it shall be present. The  
 29 associating entity is contiguous if and only if the selector is contiguous.

30 If the selector (8.1.5.1) is not permitted to appear in a variable definition context (16.6.7) or is an array  
 31 with a vector subscript, the associate name shall not appear in a variable definition context.

#### 32 8.1.5.4 Examples of the ASSOCIATE construct

##### NOTE 8.10

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))
```

**NOTE 8.10 (cont.)**

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
```

## 1 8.1.6 SELECT TYPE construct

- 2 The SELECT TYPE construct selects for execution at most one of its constituent blocks. The selection  
3 is based on the dynamic type of an expression. A name is associated with the expression (16.4, 16.5.1.6),  
4 in the same way as for the ASSOCIATE construct.

### 5 8.1.6.1 Form of the SELECT TYPE construct

```
6 R821  select-type-construct      is  select-type-stmt
7                                     [ type-guard-stmt
8                                     block ] ...
9                                     end-select-type-stmt
10 R822  select-type-stmt          is  [ select-construct-name : ] SELECT TYPE ■
11                                     ■ ( [ associate-name => ] selector )
```

12 C811 (R822) If *selector* is not a named *variable*, *associate-name* => shall appear.

13 C812 (R822) 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 C813 (R822) The *selector* in a *select-type-stmt* shall be polymorphic.

```
16 R823  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 C814 (R823) The *type-spec* or *derived-type-spec* shall specify that each length type parameter is as-  
20 sumed.

21 C815 (R823) The *type-spec* or *derived-type-spec* shall not specify a sequence derived type or a type  
22 with the BIND attribute.

23 C816 (R823) 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 C817 (R823) 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 C818 (R823) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-*



1        *guard-stmt*.

2    R824    *end-select-type-stmt*        **is**    END SELECT [ *select-construct-name* ]

3    C819    (R821) If the *select-type-stmt* of a *select-type-construct* specifies a *select-construct-name*, the  
 4        corresponding *end-select-type-stmt* shall specify the same *select-construct-name*. If the *select-*  
 5        *type-stmt* of a *select-type-construct* does not specify a *select-construct-name*, the corresponding  
 6        *end-select-type-stmt* shall not specify a *select-construct-name*. If a *type-guard-stmt* specifies a  
 7        *select-construct-name*, the corresponding *select-type-stmt* shall specify the same *select-construct-*  
 8        *name*.

9    The associate name of a SELECT TYPE construct is the *associate-name* if specified; otherwise it is the  
 10    *name* that constitutes the *selector*.

### 11    **8.1.6.2    Execution of the SELECT TYPE construct**

12    Execution of a SELECT TYPE construct whose selector is not a *variable* causes the selector expression  
 13    to be evaluated.

14    A SELECT TYPE construct selects at most one block to be executed. During execution of that block,  
 15    the associate name identifies an entity, which is associated (16.5.1.6) with the selector.

16    A TYPE IS type guard statement matches the selector if the dynamic type and kind type parameter  
 17    values of the selector are the same as those specified by the statement. A CLASS IS type guard  
 18    statement matches the selector if the dynamic type of the selector is an extension of the type specified  
 19    by the statement and the kind type parameter values specified by the statement are the same as the  
 20    corresponding type parameter values of the dynamic type of the selector.

21    The block to be executed is selected as follows.

- 22        (1)    If a TYPE IS type guard statement matches the selector, the block following that statement  
 23                is executed.
- 24        (2)    Otherwise, if exactly one CLASS IS type guard statement matches the selector, the block  
 25                following that statement is executed.
- 26        (3)    Otherwise, if several CLASS IS type guard statements match the selector, one of these  
 27                statements must specify a type that is an extension of all the types specified in the others;  
 28                the block following that statement is executed.
- 29        (4)    Otherwise, if there is a CLASS DEFAULT type guard statement, the block following that  
 30                statement is executed.

#### NOTE 8.11

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.

31    Within the block following a TYPE IS type guard statement, the associating entity (16.5.5) is not  
 32    polymorphic (4.3.1.3), has the type named in the type guard statement, and has the type parameter  
 33    values of the selector.

34    Within the block following a CLASS IS type guard statement, the associating entity is polymorphic and  
 35    has the declared type named in the type guard statement. The type parameter values of the associating  
 36    entity are the corresponding type parameter values of the selector.

37    Within the block following a CLASS DEFAULT type guard statement, the associating entity is poly-  
 38    morphic and has the same declared type as the selector. The type parameter values of the associating  
 39    entity are those of the declared type of the selector.

**NOTE 8.12**

If the declared type of the *selector* is T, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (T).

- 1 The other attributes of the associating entity are described in 8.1.5.3.
- 2 A type guard statement shall not be a branch target statement. It is permissible to branch to an
- 3 *end-select-type-stmt* only from within its SELECT TYPE construct.
- 4 **8.1.6.3 Examples of the SELECT TYPE construct**

**NOTE 8.13**

```

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 8.14**

The following example illustrates the omission of *associate-name*. It uses the declarations from Note 8.13.

```

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 8.1.7 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, an EXIT statement (8.1.7.5.4) anywhere in the DO construct may be executed to terminate the  
8 loop immediately. The current iteration of the loop may be curtailed by executing a CYCLE statement  
9 (8.1.7.5.3).

10 There are three phases in the execution of a DO construct: initiation of the loop, execution of the loop  
11 range, and termination of the loop.

12 The DO CONCURRENT construct is a DO construct whose DO statement contains the CONCURRENT  
13 keyword.

### 14 8.1.7.2 Forms of the DO construct

15 The DO construct may be written in either a block form or a nonblock form.

16 R825 *do-construct*                                **is** *block-do-construct*  
17                                                        **or** *nonblock-do-construct*

#### 18 8.1.7.2.1 Form of the block DO construct

19 R826 *block-do-construct*                        **is** *do-stmt*  
20                                                        *do-block*  
21                                                        *end-do*  
22 R827 *do-stmt*                                    **is** *label-do-stmt*  
23                                                        **or** *nonlabel-do-stmt*  
24 R828 *label-do-stmt*                            **is** [ *do-construct-name* : ] DO *label* [ *loop-control* ]  
25 R829 *nonlabel-do-stmt*                        **is** [ *do-construct-name* : ] DO [ *loop-control* ]  
26 R830 *loop-control*                            **is** [ , ] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■  
27                                                        ■ [ , *scalar-int-expr* ]  
28                                                        **or** [ , ] WHILE ( *scalar-logical-expr* )  
29                                                        **or** [ , ] CONCURRENT *forall-header*  
30 R831 *do-variable*                            **is** *scalar-int-variable-name*

31 C820 (R831) The *do-variable* shall be a variable of type integer.

32 R832 *do-block*                                    **is** *block*  
33 R833 *end-do*                                    **is** *end-do-stmt*  
34                                                        **or** *continue-stmt*  
35 R834 *end-do-stmt*                            **is** END DO [ *do-construct-name* ]

36 C821 (R826) If the *do-stmt* of a *block-do-construct* specifies a *do-construct-name*, the corresponding  
37 *end-do* shall be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a  
38 *block-do-construct* does not specify a *do-construct-name*, the corresponding *end-do* shall not  
39 specify a *do-construct-name*.

40 C822 (R826) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

41 C823 (R826) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the  
42 same *label*.

### 1 8.1.7.2.2 Form of the nonblock DO construct

2 R835 *nonblock-do-construct* is *action-term-do-construct*  
 3 or *outer-shared-do-construct*  
 4 R836 *action-term-do-construct* is *label-do-stmt*  
 5 *do-body*  
 6 *do-term-action-stmt*  
 7 R837 *do-body* is [ *execution-part-construct* ] ...  
 8 R838 *do-term-action-stmt* is *action-stmt*

9 C824 (R838) A *do-term-action-stmt* shall not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*,  
 10 a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

11 C825 (R835) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer  
 12 to the same label.

13 R839 *outer-shared-do-construct* is *label-do-stmt*  
 14 *do-body*  
 15 *shared-term-do-construct*

16 R840 *shared-term-do-construct* is *outer-shared-do-construct*  
 17 or *inner-shared-do-construct*

18 R841 *inner-shared-do-construct* is *label-do-stmt*  
 19 *do-body*  
 20 *do-term-shared-stmt*

21 R842 *do-term-shared-stmt* is *action-stmt*

22 C826 (R842) A *do-term-shared-stmt* shall not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*,  
 23 an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

24 C827 (R840) The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmts* of the *inner-shared-*  
 25 *do-construct* and *outer-shared-do-construct* shall refer to the same label.

26 The *do-term-action-stmt*, *do-term-shared-stmt*, or *shared-term-do-construct* following the *do-body* of a nonblock DO con-  
 27 struct is called the **DO termination** of that construct.

28 Within a scoping unit, all DO constructs whose DO statements refer to the same label are nonblock DO constructs, and  
 29 are said to share the statement identified by that label.

### 30 8.1.7.3 Range of the DO construct

31 The **range** of a block DO construct is the *do-block*, which shall satisfy the rules for blocks (8.1.2). In  
 32 particular, transfer of control to the interior of such a block from outside the block is prohibited. It  
 33 is permitted to branch to the *end-do* of a block DO construct only from within the range of that DO  
 34 construct.

35 The range of a nonblock DO construct consists of the *do-body* and the following DO termination. The end of such a  
 36 range is not bounded by a particular statement as for the other executable constructs (e.g., END IF); nevertheless, the  
 37 range satisfies the rules for blocks (8.1.2). Transfer of control into the *do-body* or to the DO termination from outside the  
 38 range is prohibited; in particular, it is permitted to branch to a *do-term-shared-stmt* only from within the range of the  
 39 corresponding *inner-shared-do-construct*.

### 40 8.1.7.4 Active and inactive DO constructs

41 A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only  
 42 when its DO statement is executed.

43 Once active, the DO construct becomes inactive only when it terminates (8.1.7.5.4).

### 44 8.1.7.5 Execution of a DO construct

#### 45 8.1.7.5.1 Loop initiation

46 When the DO statement is executed, the DO construct becomes active. If *loop-control* is

1      $[ , ]$  *do-variable* = *scalar-int-expr*<sub>1</sub> , *scalar-int-expr*<sub>2</sub>  $[ ,$  , *scalar-int-expr*<sub>3</sub> ]

2     the following steps are performed in sequence.

- 3       (1)   The initial parameter  $m_1$ , the terminal parameter  $m_2$ , and the incrementation parameter
- 4            $m_3$  are of type integer with the same kind type parameter as the *do-variable*. Their values
- 5           are established by evaluating *scalar-int-expr*<sub>1</sub>, *scalar-int-expr*<sub>2</sub>, and *scalar-int-expr*<sub>3</sub>, re-
- 6           spectively, including, if necessary, conversion to the kind type parameter of the *do-variable*
- 7           according to the rules for numeric conversion (Table 7.11). If *scalar-int-expr*<sub>3</sub> does not
- 8           appear,  $m_3$  has the value 1. The value of  $m_3$  shall not be zero.
- 9       (2)   The DO variable becomes defined with the value of the initial parameter  $m_1$ .
- 10      (3)   The **iteration count** is established and is the value of the expression  $(m_2 - m_1 + m_3)/m_3$ ,
- 11           unless that value is negative, in which case the iteration count is 0.

#### NOTE 8.15

The iteration count is zero whenever:

$m_1 > m_2$  and  $m_3 > 0$ , or  
 $m_1 < m_2$  and  $m_3 < 0$ .

12   If *loop-control* is omitted, no iteration count is calculated. The effect is as if a large positive iteration  
 13   count, impossible to decrement to zero, were established. If *loop-control* is  $[ , ]$  WHILE (*scalar-logical-*  
 14   *expr*), the effect is as if *loop-control* were omitted and the following statement inserted as the first  
 15   statement of the *do-block*:

16   IF ( .NOT. (*scalar-logical-expr*) ) EXIT

17   For a DO CONCURRENT construct, the values of the index variables for the iterations of the construct  
 18   are determined by the rules for the index variables of the FORALL construct (7.4.4.2.1 and 7.4.4.2.2).  
 19   The number of distinct index value combinations in the active combination of *index-name* values is the  
 20   iteration count for the construct.

21   An *index-name* in a DO CONCURRENT construct has a scope of the construct (16.4). It is a scalar  
 22   variable that has the type and type parameters that it would have if it were the name of a variable in  
 23   the scoping unit that includes the DO CONCURRENT, and this type shall be integer type; it has no  
 24   other attributes.

25   At the completion of the execution of the DO statement, the execution cycle begins.

#### 26   8.1.7.5.2   The execution cycle

27   The **execution cycle** of a DO construct that is not a DO CONCURRENT construct consists of the  
 28   following steps performed in sequence repeatedly until termination.

- 29       (1)   The iteration count, if any, is tested. If it is zero, the loop terminates and the DO construct
- 30           becomes inactive. If *loop-control* is  $[ , ]$  WHILE (*scalar-logical-expr*), the *scalar-logical-expr*
- 31           is evaluated; if the value of this expression is false, the loop terminates and the DO construct
- 32           becomes inactive. If, as a result, all of the DO constructs sharing the *do-term-shared-stmt* are inactive,
- 33           the execution of all of these constructs is complete. However, if some of the DO constructs sharing the
- 34           *do-term-shared-stmt* are active, execution continues with step (3) of the execution cycle of the active DO
- 35           construct whose DO statement was most recently executed.
- 36       (2)   If the iteration count is nonzero, the range of the loop is executed.
- 37       (3)   The iteration count, if any, is decremented by one. The DO variable, if any, is incremented
- 38           by the value of the incrementation parameter  $m_3$ .

1 Except for the incrementation of the DO variable that occurs in step (3), the DO variable shall neither  
 2 be redefined nor become undefined while the DO construct is active.

3 The range of a DO CONCURRENT construct is executed for all of the active combinations of the  
 4 *index-name* values. Each execution of the range is an **iteration**. The executions may occur in any  
 5 order.

#### J3 internal note

##### Unresolved Technical Issue 25

This definition of “iteration” for DO CONCURRENT almost certainly breaks the use of the term elsewhere throughout the standard. Making it not a definition of “iteration” would probably suffice to fix this.

#### 6 8.1.7.5.3 CYCLE statement

7 Execution of the range of the loop may be curtailed by executing a CYCLE statement from within the  
 8 range of the loop.

9 R843 *cycle-stmt* is CYCLE [ *do-construct-name* ]

10 C828 (R843) If a *cycle-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-*  
 11 *construct*; otherwise, it shall be within the range of at least one *do-construct*.

12 C829 (R843) A *cycle-stmt* shall not appear within the range of a DO CONCURRENT construct if it  
 13 belongs to an outer construct.

14 A CYCLE statement belongs to a particular DO construct. If the CYCLE statement refers to a DO  
 15 construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct  
 16 in which it appears.

17 Execution of a CYCLE statement that belongs to a DO construct that is not a DO CONCURRENT  
 18 construct causes immediate progression to step (3) of the current execution cycle of the DO construct  
 19 to which it belongs. If this construct is a nonblock DO construct, the *do-term-action-stmt* or *do-term-shared-stmt* is  
 20 not executed.

21 Execution of a CYCLE statement that belongs to a DO CONCURRENT construct completes execution  
 22 of that iteration of the construct.

23 In a block DO construct, a transfer of control to the *end-do* has the same effect as execution of a CYCLE  
 24 statement belonging to that construct. In a nonblock DO construct, transfer of control to the *do-term-action-stmt*  
 25 or *do-term-shared-stmt* causes that statement or construct itself to be executed. Unless a further transfer of control results,  
 26 step (3) of the current execution cycle of the DO construct is then executed.

#### 27 8.1.7.5.4 Loop termination

28 For a DO construct that is not a DO CONCURRENT construct, the loop terminates, and the DO  
 29 construct becomes inactive, when any of the following occurs.

- 30 (1) The iteration count is determined to be zero or the *scalar-logical-expr* is false, when tested  
 31 during step (1) of the above execution cycle.
- 32 (2) An EXIT statement belonging to the DO construct is executed.
- 33 (3) An EXIT or CYCLE statement that belongs to an outer construct and is within the range  
 34 of the DO construct is executed.
- 35 (4) Control is transferred from a statement within the range of a DO construct to a statement  
 36 that is neither the *end-do* nor within the range of the same DO construct.

- 1       (5) A RETURN statement within the range of the DO construct is executed.  
2       (6) A STOP statement anywhere in the program is executed, or the program is terminated for  
3       any other reason.
- 4 For a DO CONCURRENT construct, the loop terminates, and the DO construct becomes inactive when  
5 all of the iterations have completed execution.
- 6 When a DO construct becomes inactive, the DO variable, if any, of the DO construct retains its last  
7 defined value.

#### 8 **8.1.7.6 Restrictions on DO CONCURRENT constructs**

9 The following restrictions apply to DO CONCURRENT constructs.

- 10       • A statement in the loop range shall not cause a branch out of the construct.
- 11       • A variable that is referenced in an iteration shall either be previously defined during that iteration,  
12       or shall be defined or become undefined during any other iteration of the current execution of  
13       the construct. A variable that is defined or becomes undefined by more than one iteration of the  
14       current execution of the construct becomes undefined when the current execution of the construct  
15       terminates.
- 16       • A pointer that is referenced in an iteration either shall be previously pointer associated during that  
17       iteration, or shall not have its pointer association changed during any iteration. A pointer that has  
18       its pointer association changed in more than one iteration has a processor dependent association  
19       status when the construct terminates.
- 20       • An allocatable object that is allocated in more than one iteration shall be subsequently deallocated  
21       during the same iteration in which it was allocated. An object that is allocated or deallocated in  
22       only one iteration shall not be deallocated, allocated, referenced, defined, or become undefined in  
23       a different iteration.
- 24       • An input/output statement shall not write data to a file record or position in one iteration and  
25       read from the same record or position in a different iteration of the same execution of the construct.
- 26       • Records written by output statements in the loop range to a sequential access file appear in the  
27       file in an indeterminate order.
- 28       • Procedures referenced in the loop range shall be PURE. References to the procedures IEEE\_GET\_FLAG,  
29       IEEE\_SET\_HALTING\_MODE, and IEEE\_GET\_HALTING\_MODE from the intrinsic module IEEE\_EXCEPTIONS  
30       shall not appear in the loop range.

#### **NOTE 8.16**

The restrictions on referencing variables defined in an iteration of a DO CONCURRENT construct apply to any procedure invoked within the loop.

#### **NOTE 8.17**

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 8.18**

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 8.19**

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 8.20**

The following example behaves exactly the same as the one in Note 8.19. 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)
  . . .
  CALL SUBZ (X)
END DO
```

**NOTE 8.21**

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 will allow the compiler to generate vector gather/scatter code, unroll the loop, or parallelize the code for this loop, significantly improving performance.

```
INTEGER :: A(N,N),IND(N)
```



**NOTE 8.21 (cont.)**

```
DO CONCURRENT (I=1:N, J=1:N)
  A(IND(I),IND(J)) = A(IND(I),IND(J)) + 1
END DO
```

**NOTE 8.22**

Additional examples of DO constructs are in C.5.3.

**1 8.1.8 EXIT statement**

2 The EXIT statement provides one way of terminating a construct.

3 R844 *exit-stmt* **is** EXIT [ *construct-name* ]

4 C830 If an *exit-stmt* refers to a *construct-name*, it shall be within that construct; otherwise, it shall  
5 be within the range of at least one *do-construct*.

6 An EXIT statement belongs to a particular construct. If the EXIT statement refers to a construct name,  
7 it belongs to that construct; otherwise, it belongs to the innermost DO construct in which it appears.

8 C831 An *exit-stmt* shall not belong to a DO CONCURRENT construct, nor shall it appear within  
9 the range of a DO CONCURRENT construct if it belongs to a construct that contains that DO  
10 CONCURRENT construct.

11 When an EXIT statement that belongs to a DO construct is executed, it terminates the loop (8.1.7.5.4)  
12 and any active loops contained within the terminated loop. When an EXIT statement that belongs to  
13 a non-DO construct is executed, it terminates any active loops contained within that construct, and  
14 completes execution of that construct.

**15 8.1.9 BLOCK construct**

16 The BLOCK construct is a block which may contain declarations.

17 R845 *block-construct* **is** *block-stmt*  
18 [ *specification-part* ]  
19 *block*  
20 *end-block-stmt*

21 R846 *block-stmt* **is** [ *block-construct-name* : ] BLOCK

22 R847 *end-block-stmt* **is** END BLOCK [ *block-construct-name* ]

23 C832 (R845) If the *block-stmt* of a *block-construct* specifies a *block-construct-name*, the corresponding  
24 *end-block-stmt* shall specify the same *block-construct-name*. If the *block-stmt* does not specify  
25 a *block-construct-name*, the corresponding *end-block-stmt* shall not specify a *block-construct-*  
26 *name*.

27 Specifications in a BLOCK construct declare construct entities whose scope is that of the BLOCK  
28 construct.

29 Specification expressions in the *specification-part* of a BLOCK construct are evaluated when the BLOCK  
30 statement is executed.

31 The BLOCK construct terminates when a RETURN statement within the block is executed, or when  
32 transfer of control to a statement outside the block occurs.

### 1 8.1.10 CRITICAL construct

2 A CRITICAL construct limits execution of a block to one image at a time.

```
3 R848  critical-construct          is  critical-stmt
4                                     block
5                                     end-critical-stmt
```

```
6 R849  critical-stmt              is  [ critical-construct-name : ] CRITICAL
```

```
7 R850  end-critical-stmt          is  END CRITICAL [ critical-construct-name ]
```

8 C833 (R848) 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*.

12 C834 (R848) The *block* of a *critical-construct* shall not contain an image control statement.

13 A CRITICAL construct completes execution if the END CRITICAL statement is executed or if control  
14 is transferred to a branch target outside the block.

15 The processor shall ensure that once an image has commenced executing *block*, no other image shall  
16 commence executing it until the image has completed executing it. If image T is the next to execute the  
17 construct after image M, the segments (8.5.1) that executed before the construct on image M precede  
18 the segments that execute after the construct on image T. No image control statement shall be executed  
19 during the execution of a critical construct by the image executing the CRITICAL construct.

#### NOTE 8.23

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 8.24

The following example permits a large number of jobs to be shared among the images:

```
INTEGER :: NUM_JOBS[*], JOB

IF (THIS_IMAGE() == 1) READ(*,*) NUM_JOBS
SYNC_ALL
DO
  CRITICAL
    JOB = NUM_JOBS[1]
    NUM_JOBS[1] = JOB - 1
  END CRITICAL
  IF (JOB > 0) THEN
    ! Work on JOB
```

## NOTE 8.24 (cont.)

```

ELSE
  EXIT
END IF
END DO
SYNC_ALL

```

1 **8.2 Branching**

2 **Branching** is used to alter the normal execution sequence. A branch causes a transfer of control from  
3 one statement in a scoping unit to a labeled branch target statement in the same scoping unit. Branching  
4 may be caused by a GOTO statement, a computed GOTO statement, an arithmetic IF statement, a  
5 CALL statement that has an *alt-return-spec*, or an input/output statement that has an END= or ERR=  
6 specifier. Although procedure references and control constructs can cause transfer of control, they are  
7 not branches. A **branch target statement** is an *action-stmt*, an *associate-stmt*, an *end-associate-stmt*,  
8 an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, an *end-select-stmt*, a *select-type-stmt*, an *end-select-*  
9 *type-stmt*, a *do-stmt*, an *end-do-stmt*, a *forall-construct-stmt*, a *do-term-action-stmt*, a *do-term-shared-stmt*, or  
10 a *where-construct-stmt*.

11 **8.2.1 GO TO statement**

12 R851 *goto-stmt* is GO TO *label*

13 C835 (R851) The *label* shall be the statement label of a branch target statement that appears in the  
14 same scoping unit as the *goto-stmt*.

15 Execution of a GO TO statement causes a transfer of control so that the branch target statement  
16 identified by the label is executed next.

17 **8.2.2 Computed GO TO statement**

18 R852 *computed-goto-stmt* is GO TO ( *label-list* ) [ , ] *scalar-int-expr*

19 C836 (R852) Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same  
20 scoping unit as the *computed-goto-stmt*.

## NOTE 8.25

The same statement label may appear more than once in a label list.

21 Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If this value is  $i$  such that  
22  $1 \leq i \leq n$  where  $n$  is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed is  
23 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 continues  
24 as though a CONTINUE statement were executed.

25 **8.2.3 Arithmetic IF statement**

26 R853 *arithmetic-if-stmt* is IF ( *scalar-numeric-expr* ) *label* , *label* , *label*

27 C837 (R853) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the  
28 *arithmetic-if-stmt*.

29 C838 (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. The
- 2 branch target statement identified by the first label, the second label, or the third label is executed next depending on
- 3 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 C839 (R856) The *scalar-char-initialization-expr* shall be of default kind.

12 C840 (R856) The *scalar-int-initialization-expr* shall be of default kind.

- 13 Normal termination of execution of the program occurs on all images if a STOP statement or *end-*
- 14 *program-stmt* is executed immediately after a SYNC\_ALL statement (8.5.2) on all images. The stop
- 15 code, if any, and warnings of any exceptions that are signaling shall be made available only for image 1.

**J3 internal note****Unresolved Technical Issue 26**

What happens with execution of an *end-program-stmt* is not a proper subject for the STOP statement subclause. It is probably duplicative as well. This applies to later paragraphs as well as the above paragraph.

**J3 internal note****Unresolved Technical Issue 27**

The first sentence is ambiguous on several levels. Does it mean a particular SYNC\_ALL statement on all images?

...Or the usual "any SYNC\_ALL statement" syncs with any other?

Does it mean that a STOP or *end-program-stmt* is executed on all images?

...Or can one image decide unilaterally to normally terminate the program?

...And then, is it ok for some images to execute STOP and other to execute an *end-program-stmt*?

From the alternative (STOP statement that does not immediately follow a SYNC\_ALL statement) it seems that unilateralism might be what is intended, except that then not all cases seem to be covered in the "how many images execute a STOP" situation...

**J3 internal note****Unresolved Technical Issue 28**

"made available" is not a particularly meaningful phrase. It's use seems to contradict the (non-deleted) later paragraphs of STOP.

- 16 The processor shall ensure that once an image has commenced executing a STOP or an *end-program-stmt*
- 17 statement that does not immediately follow a SYNC\_ALL statement, no other image shall commence
- 18 executing such a statement. It causes normal termination (2.3.4) of execution of the program on that
- 19 image. The stop code, if any, and warnings of any exceptions that are signaling shall be made available

- 1 only for that image. The executions of all other images are terminated.

**J3 internal note**

Unresolved Technical Issue 29

What does this mean? The other image hangs forever? The other image is terminated immediately? How can you tell it doesn't execute a STOP (since presumably it terminates somehow anyway...). Why does this only apply to STOP not preceded by SYNC\_ALL?

2 **Editor's suggested rewrite of the above paragraphs**

- 3 Execution of a STOP statement causes normal termination of that image and termination of the program.  
 4 If there is only one image, or the STOP statement is executed immediately after a SYNC\_ALL statement  
 5 (8.5.2), the program is terminated normally (2.3.4).

- 6 If several images execute a STOP statement or *end-program-stmt*, it is processor-dependent which image's  
 7 stop code or signalling exception warning is made available, except that if a STOP statement is executed  
 8 immediately after a SYNC\_ALL statement on image 1, the stop code and signaling exceptions on image  
 9 1 are reported.

10 **End of editor's suggestion**

- 11 At the time of termination, the stop code, if any, is available in a processor-dependent manner. If  
 12 any exception (14) is signaling, the processor shall issue a warning indicating which exceptions are  
 13 signaling; this warning shall be on the unit identified by the named constant ERROR\_UNIT from the  
 14 ISO\_FORTRAN\_ENV intrinsic module (13.8.3.6).

**J3 internal note**

Unresolved Technical Issue 30

Why is image 1 special? What if image 2 does all the real work? What if every other image terminates first, before image 1 does?

Nearly everything about CAF is very symmetric with regards to images, having weird special abilities on termination of image 1 seems to be unnecessary and to have dubious value.

**J3 internal note**

Unresolved Technical Issue 31

If some image registers an "atexit" procedure, normal termination of execution is required to invoke it. Obviously this invocation ought to occur on the registering image, even if it is not image 1, but I'm less convinced that this is well-specified. We say that normal termination of execution includes the effect of executing the C exit function, one question is whether this is on every image or just one image?

This is a relatively "soft" UTI, but I think this text needs to be looked at very carefully and thought about.

**J3 internal note**

Unresolved Technical Issue 32

Exception warnings appear to be only required in F2003 for STOP, not for *end-program-stmt*. Assuming that was a deliberate decision, changing this requirement for co-arrays seems outwith the scope of the feature. Not that it might not be a good idea, but it just does not seem to have anything to do with co-arrays as such.

Perhaps this should be an F2003 interp?

- 15 It is recommended that the *stop-code* is made available by formatted output to the processor-dependent  
 16 external unit identified by the named constant ERROR\_UNIT of the ISO\_FORTRAN\_ENV intrinsic  
 17 module (13.8.3.6).

**NOTE 8.27**

If the *stop-code* is an integer, it is recommended that the value also be used as the process exit status, if the operating system supports that concept. If the integer *stop-code* is used as the process exit status, the operating system 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).

**J3 internal note****Unresolved Technical Issue 33**

This recommendation for the process exit status does not provide any help for STOP without a code, or for END PROGRAM. Perhaps “STOP” should be equivalent to “STOP 42” whereas “END PROGRAM” should be equivalent to “STOP 5”?

Not giving an exit code recommendation for “STOP charstring” is not very good either.

## 1 **8.5 Image execution control**

### 2 **8.5.1 Image control statements**

3 The execution sequence on each image is as specified in 2.3.4.

4 An **image control** statement affects the execution ordering between images. Each of the following is  
5 an image control statement:

- 6 • SYNC\_ALL statement;
- 7 • SYNC\_TEAM statement;
- 8 • SYNC\_IMAGES statement;
- 9 • SYNC\_MEMORY statement;
- 10 • NOTIFY statement;
- 11 • QUERY statement;
- 12 • ALLOCATE or DEALLOCATE statement involving a co-array;
- 13 • CRITICAL or END CRITICAL statement (8.1.10);
- 14 • OPEN statement with a TEAM= specifier;
- 15 • CLOSE statement for a file that is open with a TEAM= specifier;
- 16 • END, END BLOCK, or RETURN statement that involves an implicit deallocation of a co-array;
- 17 • CALL statement for a collective subroutine (13.1).

18 During an execution of a statement that contains more than one reference to a procedure, image control  
19 statements other than CRITICAL or END CRITICAL shall be executed in at most one of the procedures  
20 invoked.

21 On each image, the sequence of statements executed before the first image control statement, between  
22 the execution of two image control statements, or after the last image control statement is known as  
23 a **segment**. The segment executed immediately before the execution of an image control statement  
24 includes the evaluation of all expressions within the statement.

25 By execution of image control statements or user-defined ordering (8.5.6), the program can ensure  
26 that the execution of the  $i^{th}$  segment on image P,  $P_i$ , either precedes or succeeds the execution of the

- 1  $j^{th}$  segment on another image Q,  $Q_j$ . If the program does not ensure this, segments  $P_i$  and  $Q_j$  are  
 2 **unordered**; depending on the relative execution speeds of the images, some or all of the execution of  
 3 the segment  $P_i$  may take place at the same time as some or all of the execution of the segment  $Q_j$ .

**NOTE 8.28**

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 or user-defined ordering.

- 4 A scalar co-variable that is of type default integer, default logical, default real, or default bits, and has  
 5 the VOLATILE attribute (5.3.18) may be referenced during the execution of a segment that is unordered  
 6 relative to the execution of a segment in which the co-variable is defined. Otherwise,
- 7 • if a co-array variable is defined on an image in a segment, it shall not be referenced or defined in  
 8 a segment on another image unless the segments are ordered,
  - 9 • if the allocation of an allocatable subobject of a co-array or the association of a pointer subobject  
 10 of a co-array is changed on an image in a segment, that subobject shall not be referenced or defined  
 11 in a segment on another image unless the segments are ordered, and
  - 12 • if a procedure invocation on image P is in execution in segments  $P_i, P_{i+1}, \dots, P_k$  and defines a  
 13 non-co-array dummy argument, the argument associated entity shall not be referenced or defined  
 14 on another image Q in a segment  $Q_j$  unless  $Q_j$  precedes  $P_i$  or succeeds  $P_k$ .

**J3 internal note****Unresolved Technical Issue 34**

But ... re the middle item above .. you say the target of a pointer component of a co-array is always on the same image. So how can it be referenced by a remote image? Does this not mean that even non-coarrays have to be stored in remotely-accessible memory if they have the TARGET attribute?

BTW, here are what some possibilities for segment notation look like:

- (1) The  $i^{th}$  segment of P is  $P_i$ : e.g. in  $P_i, P_{i+1}, \dots, P_k$  and defines a
- (2) The  $i^{th}$  segment of P is  $S_{P,i}$ : e.g. in  $S_{P,i}, S_{P,i+1}, \dots, S_{P,k}$  and defines a
- (3) The  $i^{th}$  segment of P is  $S_{i,P}$ : e.g. in  $S_{i,P}, S_{i+1,P}, \dots, S_{k,P}$  and defines a
- (4) The  $i^{th}$  segment of P is  $S(i,P)$ : e.g. in  $S(i,P), S(i+1,P), \dots, S(k,P)$  and defines a

And here are some possibilities for the Notify/Query notation:

- (1) Count of NOTIFYs from image T to image M is  $N_{T \rightarrow M}$ ,  
 Count of successful QUERYs on image M of image T is  $Q_{M \leftarrow T}$ ,  
 e.g. until, ... ,  $N_{T \rightarrow M} > Q_{M \leftarrow T}$ .
- (2) Count of NOTIFYs from image T to image M is  $N_{T,M}$ ,  
 Count of successful QUERYs on image M of image T is  $Q_{M,T}$ ,  
 e.g. until, ... ,  $N_{T,M} > Q_{M,T}$ .
- (3) Count of NOTIFYs from image T to image M is  $n(T,M)$ ,  
 Count of successful QUERYs on image M of image T is  $b(M,T)$ ,  
 e.g. until, ... ,  $n(T,M) > b(M,T)$ .

**NOTE 8.29**

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 8.30**

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 8.31**

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**

R857 *sync-all-stmt*                   **is** SYNC\_ALL [ ( [ *sync-stat-list* ] ) ]

R858 *sync-stat*                       **is** STAT = *stat-variable*  
 11                                       **or** ERRMSG = *errmsg-variable*

C841 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 8.32**

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 8.33**

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 C842 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 invoking the intrinsic function FORM\_TEAM for the team. They do not commence  
 10 executing subsequent statements until all images in the team have executed a SYNC\_TEAM statement  
 11 for the team an equal number of times since FORM\_TEAM was invoked for the team. If images M  
 12 and T are any two members of the team, the segments that execute before the statement on image M  
 13 precede the segments that execute after the statement on image T.

14 Team synchronization is also performed by an OPEN statement with a TEAM= specifier, a CLOSE  
 15 statement for a file that is open with a TEAM= specifier, or the invocation of a collective subroutine.

**J3 internal note**

## Unresolved Technical Issue 35

It is not clear whether “the team” refers to the value gotten back from FORM\_TEAM, or the input value to FORM\_TEAM.

Yes, I know FORM\_TEAM is a “collective”, but that does not resolve this ambiguity.

Images 1-6 do

```
CALL form_team(team1,image_list)
```

```
CALL form_team(team2,image_list) ! same list as in previous statement
```

then images 1-3 do

```
SYNC_TEAM(team1)
```

whereas images 4-6 do

```
SYNC_TEAM(team2)
```

The question is, is this equivalent to them all doing

```
SYNC_TEAM(team1)
```

???

If it is, how can SYNC\_TEAM(team1) offer any advantage over SYNC\_IMAGES(image\_list)?

SYNC\_TEAM seems completely redundant.

Anyway, it needs to be made clear what the answer is.

**NOTE 8.34**

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)
```

**NOTE 8.34 (cont.)**

```

      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 C843 An *image-set* that is an *int-expr* shall be scalar or of rank one and shall not be a co-indexed  
 6 object.

**J3 internal note****Unresolved Technical Issue 36**

The restriction on *image-set* not being a co-indexed object seems to be a restriction which has no purpose and no effect.

I note that it is an expression, not a variable, so the user can just write

```
SYNC_IMAGES((x[i]%imagelist))
```

instead of

```
SYNC_IMAGES(x[i]%imagelist)
```

This kind of thing is not going to improve the reputation of the Fortran standard and the committees.

7 If *image-set* is an array expression, the value of each element shall be positive and not greater than the  
 8 number of images, and there shall be no repeated values.

9 If *image-set* is a scalar expression, its value shall be positive and not greater than the number of images.

10 An *image-set* that is an asterisk specifies all images.

11 Execution of a SYNC\_IMAGES statement performs a synchronization of the image with each of the  
 12 other images in the *image-set*. Execution on an image, M, of the segment following the SYNC\_IMAGES  
 13 statement is delayed until each other image T in the *image-set* has executed a SYNC\_IMAGES statement  
 14 with M in its *image-set* as many times as image M has executed a SYNC\_IMAGES statement with T in  
 15 the *image-set*. The segments that executed before the SYNC\_IMAGES statement on image M precede  
 16 the segments that execute after the SYNC\_IMAGES statement on image T.

**NOTE 8.35**

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

**NOTE 8.35 (cont.)**

execution in the limiting case of the number of images being equal to one.

**NOTE 8.36**

Execution of a SYNC\_IMAGES(\*) statement is not equivalent to the execution of a SYNC\_ALL statement. SYNC\_ALL causes all images to wait for each other. SYNC\_IMAGES statements are not required to specify the same image set on all the 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 8.37**

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 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 | <i>notify-stmt</i> | <b>is</b> | NOTIFY ( <i>image-set</i> [ , <i>sync-stat-list</i> ] ) |
| 3 | R864 | <i>query-stmt</i>  | <b>is</b> | QUERY ( <i>image-set</i> [ , <i>query-spec-list</i> ] ) |
| 4 | R865 | <i>query-spec</i>  | <b>is</b> | READY = <i>scalar-logical-variable</i>                  |

- 1 **or** *sync-stat*
- 2 C844 (R864) No specifier shall appear more than once in a given *query-spec-list*.
- 3 Execution on image M of a NOTIFY statement with a different image T in its *image-set* increments by
- 4 1 a record of the number of times,  $N_{M \rightarrow T}$ , image M executed such a NOTIFY statement.
- 5 A QUERY statement is **blocking** if and only if it has no READY= specifier. A QUERY statement
- 6 is **satisfied** on completion of its execution if and only if it is a blocking QUERY statement or it set
- 7 the variable specified by its READY= specifier to true. Execution on image M of a satisfied QUERY
- 8 statement with a different image T included in its image set increases by 1 a record of the number of
- 9 times,  $Q_{M \leftarrow T}$ , image M executed such a QUERY statement. This increase is made after its value has
- 10 been compared with  $N_{T \rightarrow M}$ .
- 11 If a READY= specifier appears, execution on image M of a QUERY statement causes the *scalar-logical-*
- 12 *variable* to become defined. The value is false if, for a different image T in the image set,  $N_{T \rightarrow M} \leq$
- 13  $Q_{M \leftarrow T}$ . Otherwise, the value is true.
- 14 If a READY= specifier does not appear, increasing  $Q_{M \leftarrow T}$  and completing execution of the statement
- 15 is delayed until, for each different T in the image set,  $N_{T \rightarrow M} > Q_{M \leftarrow T}$ .
- 16 A NOTIFY statement execution on image T and a satisfied QUERY statement on image M correspond
- 17 if and only if
- 18 • the NOTIFY statement's image set includes image M,
  - 19 • the QUERY statement's image set includes image T, and
  - 20 • after execution of both statements has completed,  $N_{T \rightarrow M} = Q_{M \leftarrow T}$ .
- 21 Segments on an image executed before the execution of a NOTIFY statement precede the segments on
- 22 other images that follow its corresponding QUERY statements.

**NOTE 8.38**

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
  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

```

**NOTE 8.38 (cont.)**

```
! 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 8.39**

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 8.40**

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.

- 5 All of the other image control statements include the effect of executing a SYNC\_MEMORY statement.  
6 In addition, the other image control statements cause some form of cooperation with other images for  
7 the purpose of ordering execution between images.
- 8 SYNC\_MEMORY in combination with user-written code that forces synchronization between images  
9 can provide an effective and flexible alternative to the image control statements described in 8.5.2 to  
10 8.5.5.

**J3 internal note**

Unresolved Technical Issue 37

The above paragraph contributes nothing of substance, being only an advertisement for this subfeature.

**NOTE 8.41**

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 8.42**

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
CALL EXTERNAL_SYNC()
SYNC_MEMORY
IF (IAM == 2) WRITE(*,*) X[1]

```

**NOTE 8.42 (cont.)**

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, the effect is processor dependent.

**J3 internal note****Unresolved Technical Issue 38**

This is substantially different from our existing uses of STAT=; there, any error condition that is not handled by STAT= causes termination of the program. Similarly for IOSTAT=.

The CoFortran programmer deserves no less consideration. He should not be *required* to use STAT= followed by “IF (stat/=0) STOP” to ensure his program doesn’t just go off the deep end with a processor-detected error.

This doesn’t close the door to processor extensions to return “non-fatal” errors as negative values.

- 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 8.43**

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.



## 1 9 Input/output statements

2 **Input statements** provide the means of transferring data from external media to internal storage or  
3 from an internal file to internal storage. This process is called **reading**. **Output statements** provide  
4 the means of transferring data from internal storage to external media or from internal storage to an  
5 internal file. This process is called **writing**. Some input/output statements specify that editing of the  
6 data is to be performed.

7 In addition to the statements that transfer data, there are auxiliary input/output statements to ma-  
8 nipulate the external medium, or to describe or inquire about the properties of the connection to the  
9 external medium.

10 The input/output statements are the OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, END-  
11 FILE, REWIND, FLUSH, WAIT, and INQUIRE statements.

12 The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT  
13 statement are **data transfer output statements**. The OPEN statement and the CLOSE state-  
14 ment are **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The  
15 BACKSPACE, ENDFILE, and REWIND statements are **file positioning statements**.

16 A file is composed of either a sequence of file storage units (9.2.4) or a sequence of records, which  
17 provide an extra level of organization to the file. A file composed of records is called a **record file**. A  
18 file composed of file storage units is called a **stream file**. A processor may allow a file to be viewed  
19 both as a record file and as a stream file; in this case the relationship between the file storage units when  
20 viewed as a stream file and the records when viewed as a record file is processor dependent.

21 A file is either an external file (9.2) or an internal file (9.3).

### 22 9.1 Records

23 A **record** is a sequence of values or a sequence of characters. For example, a line on a terminal is usually  
24 considered to be a record. However, a record does not necessarily correspond to a physical entity. There  
25 are three kinds of records:

- 26 (1) formatted;
- 27 (2) unformatted;
- 28 (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.”

#### 29 9.1.1 Formatted record

30 A **formatted record** consists of a sequence of characters that are capable of representation in the  
31 processor; however, a processor may prohibit some control characters (3.1) from appearing in a formatted  
32 record. The length of a formatted record is measured in characters and depends primarily on the number  
33 of characters put into the record when it is written. However, it may depend on the processor and the  
34 external medium. The length may be zero. Formatted records may be read or written only by formatted  
35 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 (9.5.2) 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 (9.7.1).

## 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 (9.4.6.7) 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 are said to **exist** for a program.  
 4 A file 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 (9.4.6.3) 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 (9.9.1.21).

- 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 9.5.3.7.

### 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 a formatted stream output statement, if no error condition occurred, the terminal point of the file  
2 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).

- 3 For formatted stream input, if an end-of-file condition occurred, the file position is not changed.
- 4 For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-record  
5 condition (9.10) occurred, the file is positioned after the record just read. If no error condition, end-of-  
6 file condition, or end-of-record condition occurred in a nonadvancing input statement, the file position  
7 is not changed. If no error condition occurred in a nonadvancing output statement, the file position is  
8 not changed.
- 9 In all other cases, the file is positioned after the record just read or written and that record becomes the  
10 preceding record.

**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 (13.8.3.7)  
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.

### 1 9.3 Internal files

2 Internal files provide a means of transferring and converting data from internal storage to internal  
3 storage.

4 An internal file is a record file with the following properties.

- 5 (1) The file is a variable of default, ASCII, or ISO 10646 character type that is not an array  
6 section with a vector subscript.
- 7 (2) A record of an internal file is a scalar character variable.
- 8 (3) If the file is a scalar character variable, it consists of a single record whose length is the same  
9 as the length of the scalar character variable. If the file is a character array, it is treated  
10 as a sequence of character array elements. Each array element, if any, is a record of the  
11 file. The ordering of the records of the file is the same as the ordering of the array elements  
12 in the array (6.2.2.2) or the array section (6.2.2.3). Every record of the file has the same  
13 length, which is the length of an array element in the array.
- 14 (4) A record of the internal file becomes defined by writing the record. If the number of  
15 characters written in a record is less than the length of the record, the remaining portion  
16 of the record is filled with blanks. The number of characters to be written shall not exceed  
17 the length of the record.
- 18 (5) A record may be read only if the record is defined.
- 19 (6) A record of an internal file may become defined (or undefined) by means other than an  
20 output statement. For example, the character variable may become defined by a character  
21 assignment statement.
- 22 (7) An internal file is always positioned at the beginning of the first record prior to data transfer,  
23 except for child data transfer statements (9.5.3.7). This record becomes the current record.
- 24 (8) The initial value of a connection mode (9.4.1) is the value that would be implied by an  
25 initial OPEN statement without the corresponding keyword.
- 26 (9) Reading and writing records shall be accomplished only by sequential access formatted  
27 input/output statements.
- 28 (10) An internal file shall not be specified as the unit in a file connection statement or a file  
29 positioning statement.

### 30 9.4 File connection

31 A **unit**, specified by an *io-unit*, provides a means for referring to a file.

32 R901	<i>io-unit</i>	<b>is</b>	<i>file-unit-number</i>
33		<b>or</b>	*
34		<b>or</b>	<i>internal-file-variable</i>
35 R902	<i>file-unit-number</i>	<b>is</b>	<i>scalar-int-expr</i>
36 R903	<i>internal-file-variable</i>	<b>is</b>	<i>char-variable</i>

37 C901 (R903) The *char-variable* shall not be an array section with a vector subscript.

38 C902 (R903) The *char-variable* shall be of type default character, ASCII character, or ISO 10646  
39 character.

40 A unit is either an external unit or an internal unit. An **external unit** is used to refer to an external  
41 file and is specified by an asterisk or a *file-unit-number*. The value of *file-unit-number* shall be non-  
42 negative, equal to one of the named constants INPUT\_UNIT, OUTPUT\_UNIT, or ERROR\_UNIT of  
43 the ISO\_FORTRAN\_ENV module (13.8.3), or a NEWUNIT value (9.4.6). An **internal unit** is used to  
44 refer to an internal file and is specified by an *internal-file-variable* or a *file-unit-number* whose value is



- 1 equal to the `unit` argument of an active derived-type input/output procedure (9.5.3.7). The value of a  
 2 *file-unit-number* shall identify a valid unit.
- 3 The external unit identified by a particular value of a *scalar-int-expr* is the same external unit in all  
 4 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.

- 5 An asterisk identifies particular processor-dependent external units that are preconnected for format-  
 6 ted sequential access (9.5.3.2). These units are also identified by unit numbers defined by the named  
 7 constants `INPUT_UNIT` and `OUTPUT_UNIT` of the `ISO_FORTRAN_ENV` module (13.8.3).
- 8 This standard identifies a processor-dependent external unit for the purpose of error reporting. This  
 9 unit shall be preconnected for sequential formatted output. The processor may define this to be the  
 10 same as the output unit identified by an asterisk. This unit is also identified by a unit number defined  
 11 by the named constant `ERROR_UNIT` of the `ISO_FORTRAN_ENV` intrinsic module.

**12 9.4.1 Connection modes**

- 13 A connection for formatted input/output has several changeable modes: the blank interpretation mode  
 14 (10.8.6), delimiter mode (10.10.2, 10.11.2.1), sign mode (10.8.4), decimal edit mode (10.8.8), I/O round-  
 15 ing mode (10.7.2.3.7), pad mode (9.5.3.4.2), and scale factor (10.8.5). A connection for unformatted  
 16 input/output has no changeable modes.

- 17 Values for the modes of a connection are established when the connection is initiated. If the connection  
 18 is initiated by an `OPEN` statement, the values are as specified, either explicitly or implicitly, by the  
 19 `OPEN` statement. If the connection is initiated other than by an `OPEN` statement (that is, if the file is  
 20 an internal file or preconnected file) the values established are those that would be implied by an initial  
 21 `OPEN` statement without the corresponding keywords.

- 22 The scale factor cannot be explicitly specified in an `OPEN` statement; it is implicitly 0.

- 23 The modes of a connection to an external file may be changed by a subsequent `OPEN` statement that  
 24 modifies the connection.

- 25 The modes of a connection may be temporarily changed by a corresponding keyword specifier in a  
 26 data transfer statement or by an edit descriptor. Keyword specifiers take effect at the beginning of  
 27 execution of the data transfer statement. Edit descriptors take effect when they are encountered in  
 28 format processing. When a data transfer statement terminates, the values for the modes are reset to the  
 29 values in effect immediately before the data transfer statement was executed.

**30 9.4.2 Unit existence**

- 31 At any given time, there is a processor-dependent set of external units that are said to exist for a  
 32 program.

- 1 All input/output statements may refer to units that exist. The CLOSE, INQUIRE, and WAIT state-  
2 ments also may refer to units that do not exist.

### 3 9.4.3 Connection of a file to a unit

- 4 An external unit has a property of being **connected** or not connected. If connected, it refers to an  
5 external file, and the connection is for a team of images. An external unit may become connected by  
6 preconnection or by the execution of an OPEN statement. The property of connection is symmetric; the  
7 unit is connected to a file if and only if the file is connected to the unit.

#### J3 internal note

##### Unresolved Technical Issue 39

The second sentence now contradicts the first sentence. Sorry guys, but the unit **\*\*\*is global\*\*\*** and only has one state, that of being connected or not. This subclause is about connection, not about whether it is formatted or unformatted, or whether there is some restriction about what image is permitted to reference it.

Not doing the edit to the second sentence would be sufficient to resolve this issue here, but later edits (under the TEAM= description) also need changing.

- 8 Every input/output statement except an OPEN, CLOSE, INQUIRE, or WAIT statement shall refer to  
9 a unit that is connected to a file and thereby make use of or affect that file.
- 10 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.

- 11 A unit shall not be connected to more than one file at the same time, and a file shall not be connected to  
12 more than one unit at the same time. However, means are provided to change the status of an external  
13 unit and to connect a unit to a different file.
- 14 This standard defines means of portable interoperability with C. C streams are described in 7.19.2 of the C  
15 International Standard. Whether a unit may be connected to a file that is also connected to a C stream  
16 is processor dependent. If the processor allows a unit to be connected to a file that is also connected to  
17 a C stream, the results of performing input/output operations on such a file are processor dependent.  
18 It is processor dependent whether the files connected to the units INPUT\_UNIT, OUTPUT\_UNIT,  
19 and ERROR\_UNIT correspond to the predefined C text streams standard input, standard output, and  
20 standard error. If a procedure defined by means of Fortran and a procedure defined by means other than  
21 Fortran perform input/output operations on the same external file, the results are processor dependent.  
22 A procedure defined by means of Fortran and a procedure defined by means other than Fortran can  
23 perform input/output operations on different external files without interference.
- 24 After an external unit has been disconnected by the execution of a CLOSE statement, it may be con-  
25 nected again within the same program to the same file or to a different file. After an external file has  
26 been disconnected by the execution of a CLOSE statement, it may be connected again within the same  
27 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.

- 28 An internal unit is always connected to the internal file designated by the variable that identifies the

1 unit.

#### NOTE 9.17

For more explanatory information on file connection properties, see C.6.5.

### 2 9.4.4 Preconnection

3 **Preconnection** means that the unit is connected to a file at the beginning of execution of the program  
4 and therefore it may be specified in input/output statements without the prior execution of an OPEN  
5 statement.

### 6 9.4.5 OPEN statement

7 An **OPEN statement** initiates or modifies the connection between an external file and a specified unit.  
8 The OPEN statement may be used to connect an existing file to a unit, create a file that is preconnected,  
9 create a file and connect it to a unit, or change certain modes of a connection between a file and a unit.

10 An external unit may be connected by an OPEN statement in any program unit of a program and, once  
11 connected, a reference to it may appear in any program unit of the program.

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 does not exist but is the same as the file to which the unit is  
16 preconnected, the modes specified by an OPEN statement become a part of the connection.

17 If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect  
18 is as if a CLOSE statement without a STATUS= specifier had been executed for the unit immediately  
19 prior to the execution of an OPEN statement.

20 If the file to be connected to the unit is the same as the file to which the unit is connected, only the  
21 specifiers for changeable modes (9.4.1) may have values different from those currently in effect. If the  
22 POSITION= specifier appears in such an OPEN statement, the value specified shall not disagree with  
23 the current position of the file. If the STATUS= specifier is included in such an OPEN statement, it shall  
24 be specified with the value OLD. Execution of such an OPEN statement causes any new values of the  
25 specifiers for changeable modes to be in effect, but does not cause any change in any of the unspecified  
26 specifiers and the position of the file is unaffected. The ERR=, IOSTAT=, and IOMSG= specifiers from  
27 any previously executed OPEN statement have no effect on any currently executed OPEN statement.

28 A STATUS= specifier with a value of OLD is always allowed when the file to be connected to the unit is  
29 the same as the file to which the unit is connected. In this case, if the status of the file was SCRATCH  
30 before execution of the OPEN statement, the file will still be deleted when the unit is closed, and the  
31 file is still considered to have a status of SCRATCH.

32 If a file is already connected to a unit, execution of an OPEN statement on that file and a different unit  
33 is not permitted.

34 R904	<i>open-stmt</i>	<b>is</b>	OPEN ( <i>connect-spec-list</i> )
35 R905	<i>connect-spec</i>	<b>is</b>	[ UNIT = ] <i>file-unit-number</i>
36		<b>or</b>	ACCESS = <i>scalar-default-char-expr</i>
37		<b>or</b>	ACTION = <i>scalar-default-char-expr</i>
38		<b>or</b>	ASYNCHRONOUS = <i>scalar-default-char-expr</i>
39		<b>or</b>	BLANK = <i>scalar-default-char-expr</i>
40		<b>or</b>	DECIMAL = <i>scalar-default-char-expr</i>

- 1                                   or DELIM = *scalar-default-char-expr*  
 2                                   or ENCODING = *scalar-default-char-expr*  
 3                                   or ERR = *label*  
 4                                   or FILE = *file-name-expr*  
 5                                   or FORM = *scalar-default-char-expr*  
 6                                   or IOMSG = *iomsg-variable*  
 7                                   or IOSTAT = *scalar-int-variable*  
 8                                   or NEWUNIT = *scalar-int-variable*  
 9                                   or PAD = *scalar-default-char-expr*  
 10                                  or POSITION = *scalar-default-char-expr*  
 11                                  or RECL = *scalar-int-expr*  
 12                                  or ROUND = *scalar-default-char-expr*  
 13                                  or SIGN = *scalar-default-char-expr*  
 14                                  or STATUS = *scalar-default-char-expr*  
 15                                  or TEAM = *image-team*  
 16 R906    *file-name-expr*           is *scalar-default-char-expr*  
 17 R907    *iomsg-variable*           is *scalar-default-char-variable*
- 18 C903    No specifier shall appear more than once in a given *connect-spec-list*.
- 19 C904    (R904) If the NEWUNIT= specifier does not appear, a *file-unit-number* shall be specified; if  
 20           the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the  
 21           *connect-spec-list*.
- 22 C905    (R904) The *label* used in the ERR= specifier shall be the statement label of a branch target  
 23           statement that appears in the same scoping unit as the OPEN statement.
- 24 C906    (R904) If a NEWUNIT= specifier appears, a *file-unit-number* shall not appear.
- 25 If the STATUS= specifier has the value NEW or REPLACE, the FILE= specifier shall appear. If the  
 26 STATUS= specifier has the value SCRATCH, the FILE= specifier shall not appear. If the STATUS=  
 27 specifier has the value OLD, the FILE= specifier shall appear unless the unit is connected and the file  
 28 connected to the unit exists.
- 29 If the NEWUNIT= specifier appears in an OPEN statement, either the FILE= specifier shall appear,  
 30 or the STATUS= specifier shall appear with a value of SCRATCH. The unit identified by a NEWUNIT  
 31 value shall not be preconnected.
- 32 A specifier that requires a *scalar-default-char-expr* may have a limited list of character values. These  
 33 values are listed for each such specifier. Any trailing blanks are ignored. The value specified is without  
 34 regard to case. Some specifiers have a default value if the specifier is omitted.
- 35 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.

#### 1 9.4.5.1 ACCESS= specifier in the OPEN statement

2 The *scalar-default-char-expr* shall evaluate to SEQUENTIAL, DIRECT, or STREAM. The ACCESS=  
3 specifier specifies the access method for the connection of the file as being sequential, direct, or stream.  
4 If this specifier is omitted, the default value is SEQUENTIAL. For an existing file, the specified access  
5 method shall be included in the set of allowed access methods for the file. For a new file, the processor  
6 creates the file with a set of allowed access methods that includes the specified method.

#### 7 9.4.5.2 ACTION= specifier in the OPEN statement

8 The *scalar-default-char-expr* shall evaluate to READ, WRITE, or READWRITE. READ specifies that  
9 the WRITE, PRINT, and ENDFILE statements shall not refer to this connection. WRITE specifies  
10 that READ statements shall not refer to this connection. READWRITE permits any input/output  
11 statements to refer to this connection. If this specifier is omitted, the default value is processor dependent.  
12 If READWRITE is included in the set of allowable actions for a file, both READ and WRITE also shall  
13 be included in the set of allowed actions for that file. For an existing file, the specified action shall be  
14 included in the set of allowed actions for the file. For a new file, the processor creates the file with a set  
15 of allowed actions that includes the specified action.

#### 16 9.4.5.3 ASYNCHRONOUS= specifier in the OPEN statement

17 The *scalar-default-char-expr* shall evaluate to YES or NO. If YES is specified, asynchronous input/output  
18 on the unit is allowed. If NO is specified, asynchronous input/output on the unit is not allowed. If this  
19 specifier is omitted, the default value is NO.

#### 20 9.4.5.4 BLANK= specifier in the OPEN statement

21 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier is permitted only  
22 for a connection for formatted input/output. It specifies the current value of the blank interpretation  
23 mode (10.8.6, 9.5.1.5) for input for this connection. This mode has no effect on output. It is a changeable  
24 mode (9.4.1). If this specifier is omitted in an OPEN statement that initiates a connection, the default  
25 value is NULL.

#### 26 9.4.5.5 DECIMAL= specifier in the OPEN statement

27 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier is per-  
28 mitted only for a connection for formatted input/output. It specifies the current value of the decimal  
29 edit mode (10.8.8, 9.5.1.6) for this connection. This is a changeable mode (9.4.1). If this specifier is  
30 omitted in an OPEN statement that initiates a connection, the default value is POINT.

#### 31 9.4.5.6 DELIM= specifier in the OPEN statement

32 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= spec-  
33 ifier is permitted only for a connection for formatted input/output. It specifies the current value of the  
34 delimiter mode (9.5.1.7) for list-directed (10.10.2) and namelist (10.11.2.1) output for the connection.  
35 This mode has no effect on input. It is a changeable mode (9.4.1). If this specifier is omitted in an  
36 OPEN statement that initiates a connection, the default value is NONE.

#### 37 9.4.5.7 ENCODING= specifier in the OPEN statement

38 The *scalar-default-char-expr* shall evaluate to UTF-8 or DEFAULT. The ENCODING= specifier is  
39 permitted only for a connection for formatted input/output. The value UTF-8 specifies that the encoding  
40 form of the file is UTF-8 as specified by ISO/IEC 10646-1:2000. Such a file is called a **Unicode** file,  
41 and all characters therein are of ISO 10646 character type. The value UTF-8 shall not be specified if  
42 the processor does not support the ISO 10646 character type. The value DEFAULT specifies that the

1 encoding form of the file is processor-dependent. If this specifier is omitted in an OPEN statement that  
2 initiates a connection, the default value is DEFAULT.

#### 3 **9.4.5.8 FILE= specifier in the OPEN statement**

4 The value of the FILE= specifier is the name of the file to be connected to the specified unit. Any trailing  
5 blanks are ignored. The *file-name-expr* shall be a name that is allowed by the processor. If this specifier  
6 is omitted and the unit is not connected to a file, the STATUS= specifier shall be specified with a value  
7 of SCRATCH; in this case, the connection is made to a processor-dependent file. The interpretation of  
8 case is processor dependent.

#### 9 **9.4.5.9 FORM= specifier in the OPEN statement**

10 The *scalar-default-char-expr* shall evaluate to FORMATTED or UNFORMATTED. The FORM= spec-  
11 ifier determines whether the file is being connected for formatted or unformatted input/output. If this  
12 specifier is omitted, the default value is UNFORMATTED if the file is being connected for direct access  
13 or stream access, and the default value is FORMATTED if the file is being connected for sequential  
14 access. For an existing file, the specified form shall be included in the set of allowed forms for the file.  
15 For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

#### 16 **9.4.6 NEWUNIT= specifier in the OPEN statement**

17 The variable is defined with a processor determined NEWUNIT value if no error occurs during the  
18 execution of the OPEN statement. If an error occurs, the processor shall not change the value of the  
19 variable.

20 A NEWUNIT value is a negative number, and shall not be equal to -1, any of the named constants  
21 ERROR\_UNIT, INPUT\_UNIT, or OUTPUT\_UNIT from the ISO\_FORTRAN\_ENV intrinsic module  
22 (13.8.3), any value used by the processor for the unit argument to a user-defined derived-type in-  
23 put/output procedure, nor any previous NEWUNIT value that identifies a file that is currently con-  
24 nected.

#### 25 **9.4.6.1 PAD= specifier in the OPEN statement**

26 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier is permitted only for a  
27 connection for formatted input/output. It specifies the current value of the pad mode (9.5.3.4.2, 9.5.1.9)  
28 for input for this connection. This mode has no effect on output. It is a changeable mode (9.4.1). If this  
29 specifier is omitted in an OPEN statement that initiates a connection, the default value is YES.

##### **NOTE 9.20**

For nondefault character types, the blank padding character is processor dependent.
-------------------------------------------------------------------------------------

#### 30 **9.4.6.2 POSITION= specifier in the OPEN statement**

31 The *scalar-default-char-expr* shall evaluate to ASIS, REWIND, or APPEND. The connection shall be  
32 for sequential or stream access. A new file is positioned at its initial point. REWIND positions an  
33 existing file at its initial point. APPEND positions an existing file such that the endfile record is the  
34 next record, if it has one. If an existing file does not have an endfile record, APPEND positions the  
35 file at its terminal point. ASIS leaves the position unchanged if the file exists and already is connected.  
36 ASIS leaves the position unspecified if the file exists but is not connected. If this specifier is omitted,  
37 the default value is ASIS.

### 1 9.4.6.3 RECL= specifier in the OPEN statement

2 The value of the RECL= specifier shall be positive. It specifies the length of each record in a file being  
 3 connected for direct access, or specifies the maximum length of a record in a file being connected for  
 4 sequential access. This specifier shall not appear when a file is being connected for stream access. This  
 5 specifier shall appear when a file is being connected for direct access. If this specifier is omitted when  
 6 a file is being connected for sequential access, the default value is processor dependent. If the file is  
 7 being connected for formatted input/output, the length is the number of characters for all records that  
 8 contain only characters of type default character. When a record contains any nondefault characters,  
 9 the appropriate value for the RECL= specifier is processor dependent. If the file is being connected for  
 10 unformatted input/output, the length is measured in file storage units. For an existing file, the value of  
 11 the RECL= specifier shall be included in the set of allowed record lengths for the file. For a new file,  
 12 the processor creates the file with a set of allowed record lengths that includes the specified value.

### 13 9.4.6.4 ROUND= specifier in the OPEN statement

14 The *scalar-default-char-expr* shall evaluate to one of UP, DOWN, ZERO, NEAREST, COMPATIBLE,  
 15 or PROCESSOR\_DEFINED. The ROUND= specifier is permitted only for a connection for formatted  
 16 input/output. It specifies the current value of the I/O rounding mode (10.7.2.3.7, 9.5.1.12) for this  
 17 connection. This is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that  
 18 initiates a connection, the I/O rounding mode is processor dependent; it shall be one of the above modes.

#### NOTE 9.21

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.

### 19 9.4.6.5 SIGN= specifier in the OPEN statement

20 The *scalar-default-char-expr* shall evaluate to one of PLUS, SUPPRESS, or PROCESSOR\_DEFINED.  
 21 The SIGN= specifier is permitted only for a connection for formatted input/output. It specifies the  
 22 current value of the sign mode (10.8.4, 9.5.1.13) for this connection. This is a changeable mode (9.4.1).  
 23 If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PRO-  
 24 CESSOR\_DEFINED.

### 25 9.4.6.6 STATUS= specifier in the OPEN statement

26 The *scalar-default-char-expr* shall evaluate to OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. If  
 27 OLD is specified, the file shall exist. If NEW is specified, the file shall not exist.

28 Successful execution of an OPEN statement with NEW specified creates the file and changes the status  
 29 to OLD. If REPLACE is specified and the file does not already exist, the file is created and the status is  
 30 changed to OLD. If REPLACE is specified and the file does exist, the file is deleted, a new file is created  
 31 with the same name, and the status is changed to OLD. If SCRATCH is specified, the file is created  
 32 and connected to the specified unit for use by the program but is deleted at the execution of a CLOSE  
 33 statement referring to the same unit or at the normal termination of the program.

#### NOTE 9.22

SCRATCH shall not be specified with a named file.

34 If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default  
 35 value is UNKNOWN.

#### 1 9.4.6.7 TEAM= specifier in the OPEN statement

- 2 The *image-team* specifies the **connect team** for the unit, which is the set of images that are permitted  
 3 to reference the unit. The team shall include the executing image. If there is no TEAM= specifier, the  
 4 connect team consists of only the executing image.”

##### J3 internal note

###### Unresolved Technical Issue 40

There is no INQUIRE(TEAM=) which is rather a glaring omission, since that particular characteristic of a unit is even more important than things like the BLANK= mode, since it determines whether an image is permitted to use the unit. After that is fixed, one might consider allowing INQUIRE to reference a unit even from an image that is not part of the team, at least to find out whether it is connected and what the team is...

It was suggested (with edits) that OPENED= should return true only for images in the connect-team. That is completely unacceptable. We cannot seriously think about breaking existing carefully-written libraries and programs that use INQUIRE to discover unused units to use for i/o. That mandates us to have OPENED= return true for any unit that is open, no matter what the TEAM= status.

- 5 All images in the connect team, and no others, shall invoke the same OPEN statement with identical  
 6 values for the *connect-specs*. There is an implicit team synchronization.
- 7 The OPEN statement connects the file on the invoking images only. If the OPEN statement does not  
 8 have a FILE= specifier and the unit is not connected to a file, the processor shall connect the same file  
 9 to the unit on all images in the connect team.

##### J3 internal note

###### Unresolved Technical Issue 41

A few problems:

- (1) The bit about connection and “invoking images” appears to be completely unnecessary, since we already say elsewhere what the effect of TEAM= is (and the effect of leaving TEAM= off).
- (2) This shares the same problem as the earlier edit where the concept of connection state is contradictory.
- (3) The bit about “shall connect the same file” appears to be completely unnecessary given the bit about the same name identifying the same file in the TEAM= case, plus the TEAM= requirement that all the *connect-specs* be the same on all the team.

- 10 If the connect team contains more than one image, the OPEN statement shall

- 11 • specify direct access or
- 12 • specify sequential access and have an ACTION= specifier that evaluates to WRITE.

#### NOTE 9.23

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.

- 13 Preconnected units and the units identified by the values INPUT\_UNIT, OUTPUT\_UNIT, and ER-  
 14 ROR\_UNIT in the intrinsic module ISO\_FORTRAN\_ENV have a connect team consisting of all the



- 1 images. If an image with index greater than one executes an input/output statement on one of these  
 2 units, it shall be a WRITE or PRINT statement.

### J3 internal note

#### Unresolved Technical Issue 42

Prohibiting INQUIRE seems ... excessive and unwarranted to say the least. Subgroup agree, but there is still a question-mark over other i/o statements. (BACKSPACE, ENDFILE, WAIT, FLUSH, CLOSE, OPEN, REWIND).

### J3 internal note

#### Unresolved Technical Issue 43

Since READ is prohibited except on image one, in what way does INPUT\_UNIT have a connect team (since no-one can do anything)? Surely it would be simpler if its connect term were image 1 only? As implied by the following note...

### NOTE 9.24

The input unit identified by \* is therefore only available on the image with index one.

## 3 9.4.7 CLOSE statement

- 4 The **CLOSE statement** is used to terminate the connection of a specified unit to an external file.
- 5 Execution of a CLOSE statement for a unit may occur in any program unit of a program and need not  
 6 occur in the same program unit as the execution of an OPEN statement referring to that unit.
- 7 Execution of a CLOSE statement performs a wait operation for any pending asynchronous data transfer  
 8 operations for the specified unit.
- 9 Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it is  
 10 permitted and affects no file.
- 11 After a unit has been disconnected by execution of a CLOSE statement, it may be connected again  
 12 within the same program, either to the same file or to a different file. After a named file has been  
 13 disconnected by execution of a CLOSE statement, it may be connected again within the same program,  
 14 either to the same unit or to a different unit, provided that the file still exists.
- 15 At normal termination of execution of a program, all units that are connected are closed. Each unit  
 16 is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in  
 17 which case the unit is closed with status DELETE.

### NOTE 9.25

The effect is as though a CLOSE statement without a STATUS= specifier were executed on each connected unit.

- |    |      |                   |           |                                          |
|----|------|-------------------|-----------|------------------------------------------|
| 18 | R908 | <i>close-stmt</i> | <b>is</b> | CLOSE ( <i>close-spec-list</i> )         |
| 19 | R909 | <i>close-spec</i> | <b>is</b> | [ UNIT = ] <i>file-unit-number</i>       |
| 20 |      |                   | <b>or</b> | IOSTAT = <i>scalar-int-variable</i>      |
| 21 |      |                   | <b>or</b> | IOMSG = <i>iomsg-variable</i>            |
| 22 |      |                   | <b>or</b> | ERR = <i>label</i>                       |
| 23 |      |                   | <b>or</b> | STATUS = <i>scalar-default-char-expr</i> |
- 24 C907 (R909) No specifier shall appear more than once in a given *close-spec-list*.
- 25 C908 (R909) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the

- 1 *file-unit-number* shall be the first item in the *close-spec-list*.
- 2 C909 (R909) The *label* used in the `ERR=` specifier shall be the statement label of a branch target  
 3 statement that appears in the same scoping unit as the `CLOSE` statement.
- 4 The *scalar-default-char-expr* has a limited list of character values. Any trailing blanks are ignored. The  
 5 value specified is without regard to case.
- 6 The `IOSTAT=`, `ERR=`, and `IOMSG=` specifiers are described in 9.10.

**NOTE 9.26**

An example of a `CLOSE` statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

**NOTE 9.27**

For more explanatory information on the `CLOSE` statement, see C.6.6.

### 7 9.4.7.1 STATUS= specifier in the CLOSE statement

- 8 The *scalar-default-char-expr* shall evaluate to `KEEP` or `DELETE`. The `STATUS=` specifier determines  
 9 the disposition of the file that is connected to the specified unit. `KEEP` shall not be specified for a file  
 10 whose status prior to execution of a `CLOSE` statement is `SCRATCH`. If `KEEP` is specified for a file that  
 11 exists, the file continues to exist after the execution of a `CLOSE` statement. If `KEEP` is specified for a  
 12 file that does not exist, the file will not exist after the execution of a `CLOSE` statement. If `DELETE` is  
 13 specified, the file will not exist after the execution of a `CLOSE` statement. If this specifier is omitted, the  
 14 default value is `KEEP`, unless the file status prior to execution of the `CLOSE` statement is `SCRATCH`,  
 15 in which case the default value is `DELETE`.

## 16 9.5 Data transfer statements

- 17 The **READ statement** is the data transfer input statement. The **WRITE statement** and the  
 18 **PRINT statement** are the data transfer output statements.

- |         |                   |           |                                                                                |
|---------|-------------------|-----------|--------------------------------------------------------------------------------|
| 19 R910 | <i>read-stmt</i>  | <b>is</b> | <code>READ ( <i>io-control-spec-list</i> ) [ <i>input-item-list</i> ]</code>   |
| 20      |                   | <b>or</b> | <code>READ format [ , <i>input-item-list</i> ]</code>                          |
| 21 R911 | <i>write-stmt</i> | <b>is</b> | <code>WRITE ( <i>io-control-spec-list</i> ) [ <i>output-item-list</i> ]</code> |
| 22 R912 | <i>print-stmt</i> | <b>is</b> | <code>PRINT format [ , <i>output-item-list</i> ]</code>                        |

**NOTE 9.28**

Examples of data transfer statements are:

```
READ (6, *) SIZE
READ 10, A, B
WRITE (6, 10) A, S, J
PRINT 10, A, S, J
10 FORMAT (2E16.3, I5)
```

### 23 9.5.1 Control information list

- 24 A **control information list** is an *io-control-spec-list*. It governs data transfer.

- 1 R913 *io-control-spec* is [ UNIT = ] *io-unit*  
2 or [ FMT = ] *format*  
3 or [ NML = ] *namelist-group-name*  
4 or ADVANCE = *scalar-default-char-expr*  
5 or ASYNCHRONOUS = *scalar-char-initialization-expr*  
6 or BLANK = *scalar-default-char-expr*  
7 or DECIMAL = *scalar-default-char-expr*  
8 or DELIM = *scalar-default-char-expr*  
9 or END = *label*  
10 or EOR = *label*  
11 or ERR = *label*  
12 or ID = *scalar-int-variable*  
13 or IOMSG = *iomsg-variable*  
14 or IOSTAT = *scalar-int-variable*  
15 or PAD = *scalar-default-char-expr*  
16 or POS = *scalar-int-expr*  
17 or REC = *scalar-int-expr*  
18 or ROUND = *scalar-default-char-expr*  
19 or SIGN = *scalar-default-char-expr*  
20 or SIZE = *scalar-int-variable*
- 21 C910 (R913) No specifier shall appear more than once in a given *io-control-spec-list*.
- 22 C911 (R913) An *io-unit* shall be specified; if the optional characters UNIT= are omitted, the *io-unit*  
23 shall be the first item in the *io-control-spec-list*.
- 24 C912 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.
- 25 C913 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.
- 26 C914 (R913) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a  
27 branch target statement that appears in the same scoping unit as the data transfer statement.
- 28 C915 (R913) A *namelist-group-name* shall be the name of a namelist group.
- 29 C916 (R913) A *namelist-group-name* shall not appear if an *input-item-list* or an *output-item-list*  
30 appears in the data transfer statement.
- 31 C917 (R913) An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.
- 32 C918 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-*  
33 *control-spec-list* and the first item shall be *io-unit*.
- 34 C919 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item  
35 in the *io-control-spec-list* and the first item shall be *io-unit*.
- 36 C920 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC=  
37 specifier or a POS= specifier.
- 38 C921 (R913) If the REC= specifier appears, an END= specifier shall not appear, a *namelist-group-*  
39 *name* shall not appear, and the *format*, if any, shall not be an asterisk.
- 40 C922 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream in-  
41 put/output statement with explicit format specification (10.2) whose control information list

1 does not contain an *internal-file-variable* as the *io-unit*.

2 C923 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.

3 C924 (R913) If a SIZE= specifier appears, an ADVANCE= specifier also shall appear.

4 C925 (R913) The *scalar-char-initialization-expr* in an ASYNCHRONOUS= specifier shall be of type  
5 default character and shall have the value YES or NO.

6 C926 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a  
7 *file-unit-number*.

8 C927 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall  
9 also appear.

10 C928 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.

11 C929 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format*  
12 or *namelist-group-name* shall also appear.

13 C930 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name*  
14 shall appear.

15 A SIZE= specifier may appear only in an input statement that contains an ADVANCE= specifier with  
16 the value NO.

17 An EOR= specifier may appear only in an input statement that contains an ADVANCE= specifier with  
18 the value NO.

19 If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a **formatted**  
20 **input/output statement**; otherwise, it is an **unformatted input/output statement**.

21 The ADVANCE=, ASYNCHRONOUS=, DECIMAL=, BLANK=, DELIM=, PAD=, SIGN=, and  
22 ROUND= specifiers have a limited list of character values. Any trailing blanks are ignored. The  
23 values specified are without regard to case.

24 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.

### NOTE 9.29

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

25 **9.5.1.1 FMT= specifier in a data transfer statement**

26 The FMT= specifier supplies a format specification or specifies list-directed formatting for a formatted  
27 input/output statement.

```

28  R914  format                is default-char-expr
29                                     or label
30                                     or *

```

31 C931 (R914) The *label* shall be the label of a FORMAT statement that appears in the same scoping  
32 unit as the statement containing the FMT= specifier.

33 The *default-char-expr* shall evaluate to a valid format specification (10.2.1 and 10.2.2).

**NOTE 9.30**

A *default-char-expr* includes a character constant.

- 1 If *default-char-expr* is an array, it is treated as if all of the elements of the array were specified in array
- 2 element order and were concatenated.
- 3 If *format* is \*, the statement is a **list-directed input/output statement**.

**NOTE 9.31**

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.

**4 9.5.1.2 NML= specifier in a data transfer statement**

- 5 The NML= specifier supplies the *namelist-group-name* (5.6). This name identifies a particular collection
- 6 of data objects on which transfer is to be performed.
- 7 If a *namelist-group-name* appears, the statement is a **namelist input/output statement**.

**8 9.5.1.3 ADVANCE= specifier in a data transfer statement**

- 9 The *scalar-default-char-expr* shall evaluate to YES or NO. The ADVANCE= specifier determines wheth-
- 10 er advancing input/output occurs for a nonchild input/output statement. If YES is specified, advancing
- 11 input/output occurs. If NO is specified, nonadvancing input/output occurs (9.2.3.1). If this specifier is
- 12 omitted from a nonchild input/output statement that allows the specifier, the default value is YES. A for-
- 13 matted child input/output statement is a nonadvancing input/output statement, and any ADVANCE=
- 14 specifier is ignored.

**15 9.5.1.4 ASYNCHRONOUS= specifier in a data transfer statement**

- 16 The ASYNCHRONOUS= specifier determines whether this input/output statement is synchronous or
- 17 asynchronous. If YES is specified, the statement and the input/output operation are said to be asyn-
- 18 chronous. If NO is specified or if the specifier is omitted, the statement and the input/output operation
- 19 are said to be synchronous.
- 20 Asynchronous input/output is permitted only for external files opened with an ASYNCHRONOUS=
- 21 specifier with the value YES in the OPEN statement.

**NOTE 9.32**

Both synchronous and asynchronous input/output are allowed for files opened with an ASYNCHRONOUS= specifier of YES. For other files, only synchronous input/output is allowed; this includes files opened with an ASYNCHRONOUS= specifier of NO, files opened without an ASYNCHRONOUS= specifier, preconnected files accessed without an OPEN statement, and internal files.

The ASYNCHRONOUS= specifier value in a data transfer statement is an initialization expression because it effects compiler optimizations and, therefore, needs to be known at compile time.

- 22 The processor may perform an asynchronous data transfer operation asynchronously, but it is not re-
- 23 quired to do so. For each external file, records and file storage units read or written by asynchronous
- 24 data transfer statements are read, written, and processed in the same order as they would have been if

1 the data transfer statements were synchronous.

2 If a variable is used in an asynchronous data transfer statement as

- 3 (1) an item in an input/output list,
- 4 (2) a group object in a namelist, or
- 5 (3) a SIZE= specifier

6 the base object of the *data-ref* is implicitly given the ASYNCHRONOUS attribute in the scoping unit  
7 of the data transfer statement. This attribute may be confirmed by explicit declaration.

8 When an asynchronous input/output statement is executed, the set of storage units specified by the  
9 item list or NML= specifier, plus the storage units specified by the SIZE= specifier, is defined to be the  
10 pending input/output storage sequence for the data transfer operation.

#### NOTE 9.33

A pending input/output storage sequence is not necessarily a contiguous set of storage units.

11 A pending input/output storage sequence **affector** is a variable of which any part is associated with a  
12 storage unit in a pending input/output storage sequence.

#### 13 9.5.1.5 BLANK= specifier in a data transfer statement

14 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier temporarily  
15 changes (9.4.1) the blank interpretation mode (10.8.6, 9.4.5.4) for the connection. If the specifier is  
16 omitted, the mode is not changed.

#### 17 9.5.1.6 DECIMAL= specifier in a data transfer statement

18 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier temporar-  
19 ily changes (9.4.1) the decimal edit mode (10.8.8, 9.4.5.5) for the connection. If the specifier is omitted,  
20 the mode is not changed.

#### 21 9.5.1.7 DELIM= specifier in a data transfer statement

22 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= spec-  
23 ifier temporarily changes (9.4.1) the delimiter mode (10.10.2, 10.11.2.1, 9.4.5.6) for the connection. If  
24 the specifier is omitted, the mode is not changed.

#### 25 9.5.1.8 ID= specifier in a data transfer statement

26 Successful execution of an asynchronous data transfer statement containing an ID= specifier causes the  
27 variable specified in the ID= specifier to become defined with a processor-dependent value. This value  
28 is referred to as the identifier of the data transfer operation. It can be used in a subsequent WAIT or  
29 INQUIRE statement to identify the particular data transfer operation.

30 If an error occurs during the execution of a data transfer statement containing an ID= specifier, the  
31 variable specified in the ID= specifier becomes undefined.

32 A child data transfer statement shall not specify the ID= specifier.

#### 33 9.5.1.9 PAD= specifier in a data transfer statement

34 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier temporarily changes  
35 (9.4.1) the pad mode (9.5.3.4.2, 9.4.6.1) for the connection. If the specifier is omitted, the mode is not  
36 changed.

#### 1 9.5.1.10 POS= specifier in a data transfer statement

2 The POS= specifier specifies the file position in file storage units. This specifier may appear in a data  
3 transfer statement only if the statement specifies a unit connected for stream access. A child data  
4 transfer statement shall not specify this specifier.

5 A processor may prohibit the use of POS= with particular files that do not have the properties necessary  
6 to support random positioning. A processor may also prohibit positioning a particular file to any  
7 position prior to its current file position if the file does not have the properties necessary to support such  
8 positioning.

#### NOTE 9.34

A file that represents connection to a device or data stream might not be positionable.
-----------------------------------------------------------------------------------------

9 If the file is connected for formatted stream access, the file position specified by POS= shall be equal to  
10 either 1 (the beginning of the file) or a value previously returned by a POS= specifier in an INQUIRE  
11 statement for the file.

#### 12 9.5.1.11 REC= specifier in a data transfer statement

13 The REC= specifier specifies the number of the record that is to be read or written. This specifier  
14 may appear only in an input/output statement that specifies a unit connected for direct access; it  
15 shall not appear in a child data transfer statement. If the control information list contains a REC=  
16 specifier, the statement is a **direct access input/output statement**. A child data transfer statement  
17 is a direct access data transfer statement if the parent is a direct access data transfer statement. Any  
18 other data transfer statement is a **sequential access input/output statement** or a **stream access**  
19 **input/output statement**, depending on whether the file connection is for sequential access or stream  
20 access.

#### 21 9.5.1.12 ROUND= specifier in a data transfer statement

22 The *scalar-default-char-expr* shall evaluate to one of the values specified in 9.4.6.4. The ROUND=  
23 specifier temporarily changes (9.4.1) the I/O rounding mode (10.7.2.3.7, 9.4.6.4) for the connection. If  
24 the specifier is omitted, the mode is not changed.

#### 25 9.5.1.13 SIGN= specifier in a data transfer statement

26 The *scalar-default-char-expr* shall evaluate to PLUS, SUPPRESS, or PROCESSOR\_DEFINED. The  
27 SIGN= specifier temporarily changes (9.4.1) the sign mode (10.8.4, 9.4.6.5) for the connection. If the  
28 specifier is omitted, the mode is not changed.

#### 29 9.5.1.14 SIZE= specifier in a data transfer statement

30 When a synchronous nonadvancing input statement terminates, the variable specified in the SIZE=  
31 specifier becomes defined with the count of the characters transferred by data edit descriptors during  
32 execution of the current input statement. Blanks inserted as padding (9.5.3.4.2) are not counted.

33 For asynchronous nonadvancing input, the storage units specified in the SIZE= specifier become defined  
34 with the count of the characters transferred when the corresponding wait operation is executed.

### 35 9.5.2 Data transfer input/output list

36 An input/output list specifies the entities whose values are transferred by a data transfer input/output  
37 statement.

- 1 R915 *input-item* **is** *variable*  
 2 **or** *io-implied-do*  
 3 R916 *output-item* **is** *expr*  
 4 **or** *io-implied-do*  
 5 R917 *io-implied-do* **is** ( *io-implied-do-object-list* , *io-implied-do-control* )  
 6 R918 *io-implied-do-object* **is** *input-item*  
 7 **or** *output-item*  
 8 R919 *io-implied-do-control* **is** *do-variable* = *scalar-int-expr* , ■  
 9 ■ *scalar-int-expr* [ , *scalar-int-expr* ]
- 10 C932 (R915) A variable that is an *input-item* shall not be a whole assumed-size array.
- 11 C933 (R915) A variable that is an *input-item* shall not be a procedure pointer.
- 12 C934 (R919) The *do-variable* shall be a named scalar variable of type integer.
- 13 C935 (R918) In an *input-item-list*, an *io-implied-do-object* shall be an *input-item*. In an *output-item-*  
 14 *list*, an *io-implied-do-object* shall be an *output-item*.
- 15 C936 (R916) An expression that is an *output-item* shall not have a value that is a procedure pointer.
- 16 An *input-item* shall not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that  
 17 contains the *input-item*.

**NOTE 9.35**

A constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but shall not appear as an input list item.

- 18 If an input item is a pointer, it shall be associated with a definable target and data are transferred from  
 19 the file to the associated target. If an output item is a pointer, it shall be associated with a target and  
 20 data are transferred from the target to the file.

**NOTE 9.36**

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).

- 21 If an input item or an output item is allocatable, it shall be allocated.
- 22 A list item shall not be polymorphic unless it is processed by a user-defined derived-type input/output  
 23 procedure (9.5.3.7).
- 24 The *do-variable* of an *io-implied-do* that is in another *io-implied-do* shall not appear as, nor be associated  
 25 with, the *do-variable* of the containing *io-implied-do*.
- 26 The following rules describing whether to expand an input/output list item are re-applied to each  
 27 expanded list item until none of the rules apply.
- 28 If an array appears as an input/output list item, it is treated as if the elements, if any, were specified in  
 29 array element order (6.2.2.2). However, no element of that array may affect the value of any expression  
 30 in the *input-item*, nor may any element appear more than once in an *input-item*.



**NOTE 9.37**

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

```

- 1 If a list item of derived type in an unformatted input/output statement is not processed by a user-defined
- 2 derived-type input/output procedure (9.5.3.7), and if any subobject of that list item would be processed
- 3 by a user-defined derived-type input/output procedure, the list item is treated as if all of the components
- 4 of the object were specified in the list in component order (4.5.4.6); those components shall be accessible
- 5 in the scoping unit containing the input/output statement and shall not be pointers or allocatable.
- 6 An effective input/output list item of derived type in an unformatted input/output statement is treated
- 7 as a single value in a processor-dependent form unless the list item or a subobject thereof is processed
- 8 by a user-defined derived-type input/output procedure (9.5.3.7).

**NOTE 9.38**

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.

- 9 If a list item of derived type in a formatted input/output statement is not processed by a user-defined
- 10 derived-type input/output procedure, that list item is treated as if all of the components of the list item
- 11 were specified in the list in component order; those components shall be accessible in the scoping unit
- 12 containing the input/output statement and shall not be pointers or allocatable.
- 13 If a derived-type list item is not treated as a list of its individual components, that list item's ultimate
- 14 components shall not have the **POINTER** or **ALLOCATABLE** attribute unless that list item is processed
- 15 by a user-defined derived-type input/output procedure.
- 16 The scalar objects resulting when a data transfer statement's list items are expanded according to the
- 17 rules in this subclause for handling array and derived-type list items are called **effective items**. Zero-
- 18 sized arrays and *io-implied-dos* with an iteration count of zero do not contribute to the effective list
- 19 items. A scalar character item of zero length is an effective list item.

**NOTE 9.39**

In a formatted input/output statement, edit descriptors are associated with effective list items, which are always scalar. The rules in 9.5.2 determine the set of effective list items corresponding to each actual list item in the statement. These rules may have to be applied repetitively until all of the effective list items are scalar items.

- 1 For an *io-implied-do*, the loop initialization and execution is the same as for a DO construct (8.1.7.5).
- 2 An input/output list shall not contain an item of nondefault character type if the input/output statement  
 3 specifies an internal file of default character type. An input/output list shall not contain an item of  
 4 nondefault character type other than ISO 10646 or ASCII character type if the input/output statement  
 5 specifies an internal file of ISO 10646 character type. An input/output list shall not contain a character  
 6 item of any character type other than ASCII character type if the input/output statement specifies an  
 7 internal file of ASCII character type.

**NOTE 9.40**

An example of an output list with an implied DO is:

```
WRITE (LP, FMT = '(10F8.2)') (LOG (A (I)), I = 1, N + 9, K), G
```

### 8 9.5.3 Execution of a data transfer input/output statement

- 9 Execution of a WRITE or PRINT statement for a file that does not exist creates the file unless an error  
 10 condition occurs.

- 11 The effect of executing a synchronous data transfer input/output statement shall be as if the following  
 12 operations were performed in the order specified.

- 13 (1) Determine the direction of data transfer.
- 14 (2) Identify the unit.
- 15 (3) Perform a wait operation for all pending input/output operations for the unit. If an error,  
 16 end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4  
 17 through 8 are skipped for the current data transfer statement.
- 18 (4) Establish the format if one is specified.
- 19 (5) If the statement is not a child data transfer statement (9.5.3.7),  
 20 (a) position the file prior to data transfer (9.2.3.2), and  
 21 (b) for formatted data transfer, set the left tab limit (10.8.1.1).
- 22 (6) Transfer data between the file and the entities specified by the input/output list (if any) or  
 23 namelist.
- 24 (7) Determine whether an error, end-of-file, or end-of-record condition has occurred.
- 25 (8) Position the file after data transfer (9.2.3.3) unless the statement is a child data transfer  
 26 statement (9.5.3.7).
- 27 (9) Cause any variable specified in a SIZE= specifier to become defined.
- 28 (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified  
 29 in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

- 30 The effect of executing an asynchronous data transfer input/output statement shall be as if the following  
 31 operations were performed in the order specified.

- 32 (1) Determine the direction of data transfer.
- 33 (2) Identify the unit.
- 34 (3) Establish the format if one is specified.
- 35 (4) Position the file prior to data transfer (9.2.3.2) and, for formatted data transfer, set the left  
 36 tab limit (10.8.1.1).
- 37 (5) Establish the set of storage units identified by the input/output list. For a READ statement,  
 38 this might require some or all of the data in the file to be read if an input variable is used  
 39 as a *scalar-int-expr* in an *io-implied-do-control* in the input/output list, as a *subscript*,  
 40 *substring-range*, *stride*, or is otherwise referenced.

- (6) Initiate an asynchronous data transfer between the file and the entities specified by the input/output list (if any) or namelist. The asynchronous data transfer may complete (and an error, end-of-file, or end-of-record condition may occur) during the execution of this data transfer statement or during a later wait operation.
- (7) Determine whether an error, end-of-file, or end-of-record condition has occurred. The conditions may occur during the execution of this data transfer statement or during the corresponding wait operation, but not both.  
Also, any of these conditions that would have occurred during the corresponding wait operation for a previously pending data transfer operation that does not have an ID= specifier may occur during the execution of this data transfer statement.
- (8) Position the file as if the data transfer had finished (9.2.3.3).
- (9) Cause any variable specified in a SIZE= specifier to become undefined.
- (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

For an asynchronous data transfer statement, the data transfers may occur during execution of the statement, during execution of the corresponding wait operation, or anywhere between. The data transfer operation is considered to be pending until a corresponding wait operation is performed.

For asynchronous output, a pending input/output storage sequence affector (9.5.1.4) shall not be redefined, become undefined, or have its pointer association status changed.

For asynchronous input, a pending input/output storage sequence affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed.

Error, end-of-file, and end-of-record conditions in an asynchronous data transfer operation may occur during execution of either the data transfer statement or the corresponding wait operation. If an ID= specifier does not appear in the initiating data transfer statement, the conditions may occur during the execution of any subsequent data transfer or wait operation for the same unit. When a condition occurs for a previously executed asynchronous data transfer statement, a wait operation is performed for all pending data transfer operations on that unit. When a condition occurs during a subsequent statement, any actions specified by IOSTAT=, IOMSG=, ERR=, END=, and EOR= specifiers for that statement are taken.

#### NOTE 9.41

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.

#### 9.5.3.1 Direction of data transfer

Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if any, or specified within the file itself for namelist input. Execution of a WRITE or PRINT statement causes values to be transferred to a file from the entities specified by the output list and format specification, if any, or by the *namelist-group-name* for namelist output.

#### 9.5.3.2 Identifying a unit

A data transfer input/output statement that contains an input/output control list includes a UNIT= specifier that identifies an external or internal unit. A READ statement that does not contain an input/output control list specifies a particular processor-dependent unit, which is the same as the unit

- 1 identified by \* in a READ statement that contains an input/output control list. The PRINT statement  
 2 specifies some other processor-dependent unit, which is the same as the unit identified by \* in a WRITE  
 3 statement. Thus, each data transfer input/output statement identifies an external or internal unit.
- 4 The unit identified by a data transfer input/output statement shall be connected to a file when execution  
 5 of the statement begins.

**NOTE 9.42**

The file may be preconnected.

**6 9.5.3.3 Establishing a format**

- 7 If the input/output control list contains \* as a format, list-directed formatting is established. If *namelist-*  
 8 *group-name* appears, namelist formatting is established. If no *format* or *namelist-group-name* is spec-  
 9 ified, unformatted data transfer is established. Otherwise, the format specification identified by the  
 10 FMT= specifier is established.
- 11 On output, if an internal file has been specified, a format specification that is in the file or is associated  
 12 with the file shall not be specified.

**13 9.5.3.4 Data transfer**

- 14 Data are transferred between the file and the entities specified by the input/output list or namelist.  
 15 The list items are processed in the order of the input/output list for all data transfer input/output  
 16 statements except namelist formatted data transfer statements. The list items for a namelist input  
 17 statement are processed in the order of the entities specified within the input records. The list items  
 18 for a namelist output statement are processed in the order in which the variables are specified in the  
 19 *namelist-group-object-list*. Effective items are derived from the input/output list items as described in  
 20 9.5.2.
- 21 All values needed to determine which entities are specified by an input/output list item are determined  
 22 at the beginning of the processing of that item.
- 23 All values are transmitted to or from the entities specified by a list item prior to the processing of any  
 24 succeeding list item for all data transfer input/output statements.

**NOTE 9.43**

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.

- 25 All values following the *name=* part of the namelist entity (10.11) within the input records are transmit-  
 26 ted to the matching entity specified in the *namelist-group-object-list* prior to processing any succeeding  
 27 entity within the input record for namelist input statements. If an entity is specified more than once  
 28 within the input record during a namelist formatted data transfer input statement, the last occurrence  
 29 of the entity specifies the value or values to be used for that entity.
- 30 An input list item, or an entity associated with it, shall not contain any portion of an established format  
 31 specification.
- 32 If the input/output item is a pointer, data are transferred between the file and the associated target.
- 33 If an internal file has been specified, an input/output list item shall not be in the file or associated with

1 the file.

**NOTE 9.44**

The file is a data object.

2 A DO variable becomes defined and its iteration count established at the beginning of processing of the  
3 items that constitute the range of an *io-implied-do*.

4 On output, every entity whose value is to be transferred shall be defined.

**5 9.5.3.4.1 Unformatted data transfer**

6 During unformatted data transfer, data are transferred without editing between the file and the entities  
7 specified by the input/output list. If the file is connected for sequential or direct access, exactly one  
8 record is read or written.

9 Objects of intrinsic or derived types may be transferred by means of an unformatted data transfer  
10 statement.

11 A value in the file is stored in a contiguous sequence of file storage units, beginning with the file storage  
12 unit immediately following the current file position.

13 After each value is transferred, the current file position is moved to a point immediately after the last  
14 file storage unit of the value.

15 On input from a file connected for sequential or direct access, the number of file storage units required  
16 by the input list shall be less than or equal to the number of file storage units in the record.

17 On input, if the file storage units transferred do not contain a value with the same type and type  
18 parameters as the input list entity, then the resulting value of the entity is processor-dependent except  
19 in the following cases.

- 20 (1) A complex list entity may correspond to two real values of the same kind stored in the file,  
21 or vice-versa.
- 22 (2) A default character list entity of length  $n$  may correspond to  $n$  default characters stored in  
23 the file, regardless of the length parameters of the entities that were written to these storage  
24 units of the file. If the file is connected for stream input, the characters may have been  
25 written by formatted stream output.

26 On output to a file connected for unformatted direct access, the output list shall not specify more values  
27 than can fit into the record. If the file is connected for direct access and the values specified by the  
28 output list do not fill the record, the remainder of the record is undefined.

29 If the file is connected for unformatted sequential access, the record is created with a length sufficient  
30 to hold the values from the output list. This length shall be one of the set of allowed record lengths for  
31 the file and shall not exceed the value specified in the RECL= specifier, if any, of the OPEN statement  
32 that established the connection.

33 If the file is not connected for unformatted input/output, unformatted data transfer is prohibited.

34 The unit specified shall be an external unit.

**35 9.5.3.4.2 Formatted data transfer**

36 During formatted data transfer, data are transferred with editing between the file and the entities  
37 specified by the input/output list or by the *namelist-group-name*. Format control is initiated and editing  
38 is performed as described in Clause 10.

- 1 The current record and possibly additional records are read or written.
- 2 Values may be transferred by means of a formatted data transfer statement to or from objects of intrinsic  
3 or derived types. In the latter case, the transfer is in the form of values of intrinsic types to or from the  
4 components of intrinsic types that ultimately comprise these structured objects unless the derived-type  
5 list item is processed by a user-defined derived-type input/output procedure (9.5.3.7).
- 6 If the file is not connected for formatted input/output, formatted data transfer is prohibited.
- 7 During advancing input when the pad mode has the value NO, the input list and format specification  
8 shall not require more characters from the record than the record contains.
- 9 During advancing input when the pad mode has the value YES, blank characters are supplied by the  
10 processor if the input list and format specification require more characters from the record than the  
11 record contains.
- 12 During nonadvancing input when the pad mode has the value NO, an end-of-record condition (9.10)  
13 occurs if the input list and format specification require more characters from the record than the record  
14 contains, and the record is complete (9.2.2.3). If the record is incomplete, an end-of-file condition occurs  
15 instead of an end-of-record condition.
- 16 During nonadvancing input when the pad mode has the value YES, blank characters are supplied by the  
17 processor if an input item and its corresponding data edit descriptor require more characters from the  
18 record than the record contains. If the record is incomplete, an end-of-file condition occurs; otherwise  
19 an end-of-record condition occurs.
- 20 If the file is connected for direct access, the record number is increased by one as each succeeding record  
21 is read or written.
- 22 On output, if the file is connected for direct access or is an internal file and the characters specified by  
23 the output list and format do not fill a record, blank characters are added to fill the record.
- 24 On output, the output list and format specification shall not specify more characters for a record than  
25 have been specified by a RECL= specifier in the OPEN statement or the record length of an internal  
26 file.

#### 27 **9.5.3.5 List-directed formatting**

28 If list-directed formatting has been established, editing is performed as described in 10.10.

#### 29 **9.5.3.6 Namelist formatting**

30 If namelist formatting has been established, editing is performed as described in 10.11.

31 Every allocatable *namelist-group-object* in the namelist group shall be allocated and every *namelist-*  
32 *group-object* that is a pointer shall be associated with a target. If a *namelist-group-object* is polymorphic  
33 or has an ultimate component that is allocatable or a pointer, that object shall be processed by a user-  
34 defined derived-type input/output procedure (9.5.3.7).

#### 35 **9.5.3.7 User-defined derived-type input/output**

36 User-defined derived-type input/output procedures allow a program to override the default handling of  
37 derived-type objects and values in data transfer input/output statements described in 9.5.2.

38 A user-defined derived-type input/output procedure is a procedure accessible by a *dtio-generic-spec*  
39 (12.4.2.1). A particular user-defined derived-type input/output procedure is selected as described in  
40 9.5.3.7.3.

### 1 9.5.3.7.1 Executing user-defined derived-type input/output data transfers

2 If a derived-type input/output procedure is selected as specified in 9.5.3.7.3, the processor shall call the se-  
 3 lected user-defined derived-type input/output procedure for any appropriate data transfer input/output  
 4 statements executed in that scoping unit. The user-defined derived-type input/output procedure controls  
 5 the actual data transfer operations for the derived-type list item.

6 A data transfer statement that includes a derived-type list item and that causes a user-defined derived-  
 7 type input/output procedure to be invoked is called a **parent data transfer statement**. A data  
 8 transfer statement that is executed while a parent data transfer statement is being processed and that  
 9 specifies the unit passed into a user-defined derived-type input/output procedure is called a **child data**  
 10 **transfer statement**.

#### NOTE 9.45

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 9.46

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.

11 A child data transfer statement is processed differently from a nonchild data transfer statement in the  
 12 following ways.

- 13 • Executing a child data transfer statement does not position the file prior to data transfer.
- 14 • An unformatted child data transfer statement does not position the file after data transfer is  
 15 complete.
- 16 • Any ADVANCE= specifier in a child input/output statement is ignored.

### 17 9.5.3.7.2 User-defined derived-type input/output procedures

18 For a particular derived type and a particular set of kind type parameter values, there are four possible  
 19 sets of characteristics for user-defined derived-type input/output procedures; one each for formatted  
 20 input, formatted output, unformatted input, and unformatted output. The user need not supply all four  
 21 procedures. The procedures are specified to be used for derived-type input/output by interface blocks  
 22 (12.4.2.1) or by generic bindings (4.5.5), with a *dtio-generic-spec* (R1208).

23 In the four interfaces, which specify the characteristics of user-defined procedures for derived-type in-  
 24 put/output, the following syntax term is used:

25 R920 *dtv-type-spec*                      **is** TYPE( *derived-type-spec* )  
 26                                              **or** CLASS( *derived-type-spec* )

27 C937 (R920) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used;

1 otherwise, the TYPE keyword shall be used.

2 C938 (R920) All length type parameters of *derived-type-spec* shall be assumed.

3 If the *dtio-generic-spec* is READ (FORMATTED), the characteristics shall be the same as those specified  
4 by the following interface:

```

5      SUBROUTINE my_read_routine_formatted           &
6              (dtv,                                &
7              unit,                                  &
8              iotype, v_list,                        &
9              iostat, iomsg)
10     ! the derived-type value/variable
11     dtv-type-spec, INTENT(INOUT) :: dtv
12     INTEGER, INTENT(IN) :: unit ! unit number
13     ! the edit descriptor string
14     CHARACTER (LEN=*), INTENT(IN) :: iotype
15     INTEGER, INTENT(IN) :: v_list(:)
16     INTEGER, INTENT(OUT) :: iostat
17     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
18     END

```

19 If the *dtio-generic-spec* is READ (UNFORMATTED), the characteristics shall be the same as those  
20 specified by the following interface:

```

21     SUBROUTINE my_read_routine_unformatted         &
22             (dtv,                                  &
23             unit,                                    &
24             iostat, iomsg)
25     ! the derived-type value/variable
26     dtv-type-spec, INTENT(INOUT) :: dtv
27     INTEGER, INTENT(IN) :: unit
28     INTEGER, INTENT(OUT) :: iostat
29     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
30     END

```

31 If the *dtio-generic-spec* is WRITE (FORMATTED), the characteristics shall be the same as those spec-  
32 ified by the following interface:

```

33     SUBROUTINE my_write_routine_formatted          &
34             (dtv,                                    &
35             unit,                                    &
36             iotype, v_list,                          &
37             iostat, iomsg)
38     ! the derived-type value/variable
39     dtv-type-spec, INTENT(IN) :: dtv
40     INTEGER, INTENT(IN) :: unit
41     ! the edit descriptor string
42     CHARACTER (LEN=*), INTENT(IN) :: iotype
43     INTEGER, INTENT(IN) :: v_list(:)
44     INTEGER, INTENT(OUT) :: iostat
45     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
46     END

```



- 1 If the *dtio-generic-spec* is WRITE (UNFORMATTED), the characteristics shall be the same as those  
 2 specified by the following interface:

```

3      SUBROUTINE my_write_routine_unformatted      &
4              (dtv,                                &
5              unit,                                &
6              iostat, iomsg)
7      ! the derived-type value/variable
8      dtv-type-spec, INTENT(IN) :: dtv
9      INTEGER, INTENT(IN) :: unit
10     INTEGER, INTENT(OUT) :: iostat
11     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
12     END
```

- 13 The actual specific procedure names (the *my....routine....* procedure names above) are not signifi-  
 14 cant. In the discussion here and elsewhere, the dummy arguments in these interfaces are referred by the  
 15 names given above; the names are, however, arbitrary.

- 16 When a user-defined derived-type input/output procedure is invoked, the processor shall pass a *unit*  
 17 argument that has a value as follows.

- 18 • If the parent data transfer statement uses a *file-unit-number*, the value of the *unit* argument shall  
 19 be that of the *file-unit-number*.
- 20 • If the parent data transfer statement is a WRITE statement with an asterisk unit or a PRINT  
 21 statement, the *unit* argument shall have the same value as the OUTPUT\_UNIT named constant  
 22 of the ISO\_FORTRAN\_ENV intrinsic module (13.8.3).
- 23 • If the parent data transfer statement is a READ statement with an asterisk unit or a READ  
 24 statement without an *io-control-spec-list*, the *unit* argument shall have the same value as the  
 25 INPUT\_UNIT named constant of the ISO\_FORTRAN\_ENV intrinsic module (13.8.3).
- 26 • Otherwise the parent data transfer statement must access an internal file, in which case the *unit*  
 27 argument shall have a processor-dependent negative value.

#### NOTE 9.47

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 (13.8.3).

- 28 For formatted data transfer, the processor shall pass an *iotype* argument that has the value

- 29 • “LISTDIRECTED” if the parent data transfer statement specified list directed formatting,
- 30 • “NAMELIST” if the parent data transfer statement specified namelist formatting, or
- 31 • “DT” concatenated with the *char-literal-constant*, if any, of the edit descriptor, if the parent data  
 32 transfer statement contained a format specification and the list item’s corresponding edit descriptor  
 33 was a DT edit descriptor.

- 1 If the parent data transfer statement is a READ statement, the `dtv` dummy argument is argument
- 2 associated with the effective list item that caused the user-defined derived-type input procedure to be
- 3 invoked, as if the effective list item were an actual argument in this procedure reference (2.5.6).
- 4 If the parent data transfer statement is a WRITE or PRINT statement, the processor shall provide the
- 5 value of the effective list item in the `dtv` dummy argument.
- 6 If the *v-list* of the edit descriptor appears in the parent data transfer statement, the processor shall
- 7 provide the values from it in the `v_list` dummy argument, with the same number of elements in the
- 8 same order as *v-list*. If there is no *v-list* in the edit descriptor or if the data transfer statement specifies
- 9 list-directed or namelist formatting, the processor shall provide `v_list` as a zero-sized array.

**NOTE 9.48**

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.

- 10 The `iostat` argument is used to report whether an error, end-of-record, or end-of-file condition (9.10)
- 11 occurs. If an error condition occurs, the user-defined derived-type input/output procedure shall assign
- 12 a positive value to the `iostat` argument. Otherwise, if an end-of-file condition occurs, the user-defined
- 13 derived-type input procedure shall assign the value of the named constant `IOSTAT_END` (13.8.3.10)
- 14 to the `iostat` argument. Otherwise, if an end-of-record condition occurs, the user-defined derived-
- 15 type input procedure shall assign the value of the named constant `IOSTAT_EOR` (13.8.3.11) to `iostat`.
- 16 Otherwise, the user-defined derived-type input/output procedure shall assign the value zero to the
- 17 `iostat` argument.
- 18 If the user-defined derived-type input/output procedure returns a nonzero value for the `iostat` argument,
- 19 the procedure shall also return an explanatory message in the `iomsg` argument. Otherwise, the procedure
- 20 shall not change the value of the `iomsg` argument.

**NOTE 9.49**

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.

- 21 If the `iostat` argument of the user-defined derived-type input/output procedure has a nonzero value
- 22 when that procedure returns, and the processor therefore terminates execution of the program as de-
- 23 scribed in 9.10, the processor shall make the value of the `iomsg` argument available in a processor-
- 24 dependent manner.
- 25 When a parent READ statement is active, an input/output statement shall not read from any external
- 26 unit other than the one specified by the dummy argument `unit` and shall not perform output to any
- 27 external unit.
- 28 When a parent WRITE or PRINT statement is active, an input/output statement shall not perform
- 29 output to any external unit other than the one specified by the dummy argument `unit` and shall not
- 30 read from any external unit.
- 31 When a parent data transfer statement is active, a data transfer statement that specifies an internal file
- 32 is permitted.
- 33 OPEN, CLOSE, BACKSPACE, ENDFILE, and REWIND statements shall not be executed while a
- 34 parent data transfer statement is active.
- 35 A user-defined derived-type input/output procedure may use a FORMAT with a DT edit descriptor for
- 36 handling a component of the derived type that is itself of a derived type. A child data transfer statement

- 1 that is a list directed or namelist input/output statement may contain a list item of derived type.
- 2 Because a child data transfer statement does not position the file prior to data transfer, the child data
- 3 transfer statement starts transferring data from where the file was positioned by the parent data transfer
- 4 statement's most recently processed effective list item or record positioning edit descriptor. This is not
- 5 necessarily at the beginning of a record.
- 6 A record positioning edit descriptor, such as TL and TR, used on **unit** by a child data transfer statement
- 7 shall not cause the record position to be positioned before the record position at the time the user-defined
- 8 derived-type input/output procedure was invoked.

**NOTE 9.50**

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.

- 9 Neither a parent nor child data transfer statement shall be asynchronous.
- 10 A user-defined derived-type input/output procedure, and any procedures invoked therefrom, shall not
- 11 define, nor cause to become undefined, any storage location referenced by any input/output list item,
- 12 the corresponding format, or any specifier in any active parent data transfer statement, except through
- 13 the **dtv** argument.

**NOTE 9.51**

A child data transfer statement shall not specify the ID=, POS=, or REC= specifiers in an input/output control list.

**NOTE 9.52**

A simple example of derived type formatted output follows. The derived type variable **chairman** has two components. The type and an associated write formatted procedure are defined in a module so as to be accessible from wherever they might be needed. It would also be possible to check that **iotype** indeed has the value 'DT' and to set **iostat** and **iomsg** accordingly.

```

MODULE p

  TYPE :: person
    CHARACTER (LEN=20) :: name
    INTEGER :: age
  CONTAINS
    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

```

**NOTE 9.52 (cont.)**

```

    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 9.53**

In the following example, the variables of the derived type `node` form a linked list, with a single value at each node. The subroutine `pwf` is used to write the values in the list, one per line.

```

MODULE p

    TYPE node
        INTEGER :: value = 0
        TYPE (NODE), POINTER :: next_node => NULL ( )
    CONTAINS
        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
    
```

**NOTE 9.53 (cont.)**

```

      IF (ASSOCIATED(dtv%next_node)) WRITE(unit, '(dt)', IOSTAT=iostat) dtv%next_node
    END SUBROUTINE pwf

END MODULE p

```

**1 9.5.3.7.3 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 9.5.3.7, and has a specific interface whose *dtv* argument  
 5 is compatible with the effective item according to the rules for argument association in 12.5.1.4.

6 When an effective item (9.5.2) 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  
 22 interface in the interface block. This reference shall not be to a dummy procedure that is not present,  
 23 or to a disassociated procedure pointer.

**24 9.5.4 Termination of data transfer statements**

25 Termination of an input/output data transfer statement occurs when

- 26 (1) format processing encounters a data edit descriptor and there are no remaining elements in  
 27 the *input-item-list* or *output-item-list*,
- 28 (2) unformatted or list-directed data transfer exhausts the *input-item-list* or *output-item-list*,
- 29 (3) namelist output exhausts the *namelist-group-object-list*,
- 30 (4) an error condition occurs,
- 31 (5) an end-of-file condition occurs,
- 32 (6) a slash (/) is encountered as a value separator (10.10, 10.11) in the record being read during  
 33 list-directed or namelist input, or
- 34 (7) an end-of-record condition occurs during execution of a nonadvancing input statement  
 35 (9.10).

## 1 9.6 Waiting on pending data transfer

2 Execution of an asynchronous data transfer statement in which neither an error, end-of-record, nor end-  
 3 of-file condition occurs initiates a pending data transfer operation. There may be multiple pending data  
 4 transfer operations for the same or multiple units simultaneously. A pending data transfer operation  
 5 remains pending until a corresponding wait operation is performed. A wait operation may be performed  
 6 by a WAIT, INQUIRE, CLOSE, or file positioning statement.

### 7 9.6.1 WAIT statement

8 A WAIT statement performs a wait operation for specified pending asynchronous data transfer opera-  
 9 tions.

#### NOTE 9.54

The CLOSE, INQUIRE, and file positioning statements may also perform wait operations.

10	R921	<i>wait-stmt</i>	<b>is</b>	WAIT ( <i>wait-spec-list</i> )
11	R922	<i>wait-spec</i>	<b>is</b>	[ UNIT = ] <i>file-unit-number</i>
12			<b>or</b>	END = <i>label</i>
13			<b>or</b>	EOR = <i>label</i>
14			<b>or</b>	ERR = <i>label</i>
15			<b>or</b>	ID = <i>scalar-int-expr</i>
16			<b>or</b>	IOMSG = <i>iomsg-variable</i>
17			<b>or</b>	IOSTAT = <i>scalar-int-variable</i>

18 C939 (R922) No specifier shall appear more than once in a given *wait-spec-list*.

19 C940 (R922) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the  
 20 *file-unit-number* shall be the first item in the *wait-spec-list*.

21 C941 (R922) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a  
 22 branch target statement that appears in the same scoping unit as the WAIT statement.

23 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.

24 The value of the expression specified in the ID= specifier shall be the identifier of a pending data transfer  
 25 operation for the specified unit. If the ID= specifier appears, a wait operation for the specified data  
 26 transfer operation is performed. If the ID= specifier is omitted, wait operations for all pending data  
 27 transfers for the specified unit are performed.

28 Execution of a WAIT statement specifying a unit that does not exist, has no file connected to it, or was  
 29 not opened for asynchronous input/output is permitted, provided that the WAIT statement has no ID=  
 30 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.

### 31 9.6.2 Wait operation

32 A wait operation completes the processing of a pending data transfer operation. Each wait operation  
 33 completes only a single data transfer operation, although a single statement may perform multiple wait  
 34 operations.

35 If the actual data transfer is not yet complete, the wait operation first waits for its completion. If the

- 1 data transfer operation is an input operation that completed without error, the storage units of the  
 2 input/output storage sequence then become defined with the values as described in 9.5.1.14 and 9.5.3.4.
- 3 If any error, end-of-file, or end-of-record conditions occur, the applicable actions specified by the IO-  
 4 STAT=, IOMSG=, ERR=, END=, and EOR= specifiers of the statement that performs the wait oper-  
 5 ation are taken.
- 6 If an error or end-of-file condition occurs during a wait operation for a unit, the processor performs a  
 7 wait operation for all pending data transfer operations for that unit.

**NOTE 9.56**

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.

- 8 After completion of the wait operation, the data transfer operation and its input/output storage sequence  
 9 are no longer considered to be pending.

## 10 9.7 File positioning statements

- |         |                       |                                                   |
|---------|-----------------------|---------------------------------------------------|
| 11 R923 | <i>backspace-stmt</i> | <b>is</b> BACKSPACE <i>file-unit-number</i>       |
| 12      |                       | <b>or</b> BACKSPACE ( <i>position-spec-list</i> ) |
| 13 R924 | <i>endfile-stmt</i>   | <b>is</b> ENDFILE <i>file-unit-number</i>         |
| 14      |                       | <b>or</b> ENDFILE ( <i>position-spec-list</i> )   |
| 15 R925 | <i>rewind-stmt</i>    | <b>is</b> REWIND <i>file-unit-number</i>          |
| 16      |                       | <b>or</b> REWIND ( <i>position-spec-list</i> )    |

17 A file that is connected for direct access shall not be referred to by a BACKSPACE, ENDFILE, or  
 18 REWIND statement. A file that is connected for unformatted stream access shall not be referred to by a  
 19 BACKSPACE statement. A file that is connected with an ACTION= specifier having the value READ  
 20 shall not be referred to by an ENDFILE statement.

- |         |                      |                                               |
|---------|----------------------|-----------------------------------------------|
| 21 R926 | <i>position-spec</i> | <b>is</b> [ UNIT = ] <i>file-unit-number</i>  |
| 22      |                      | <b>or</b> IOMSG = <i>iomsg-variable</i>       |
| 23      |                      | <b>or</b> IOSTAT = <i>scalar-int-variable</i> |
| 24      |                      | <b>or</b> ERR = <i>label</i>                  |

25 C942 (R926) No specifier shall appear more than once in a given *position-spec-list*.

26 C943 (R926) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the  
 27 *file-unit-number* shall be the first item in the *position-spec-list*.

28 C944 (R926) The *label* in the ERR= specifier shall be the statement label of a branch target statement  
 29 that appears in the same scoping unit as the file positioning statement.

30 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

31 Execution of a file positioning statement performs a wait operation for all pending asynchronous data  
 32 transfer operations for the specified unit.

### 1 9.7.1 BACKSPACE statement

- 2 Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned  
3 before the current record if there is a current record, or before the preceding record if there is no current  
4 record. If the file is at its initial point, the position of the file is not changed.

#### NOTE 9.57

If the preceding record is an endfile record, the file is positioned before the endfile record.

- 5 If a BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before  
6 the record that precedes the endfile record.
- 7 Backspacing a file that is connected but does not exist is prohibited.
- 8 Backspacing over records written using list-directed or namelist formatting is prohibited.
- 9 A BACKSPACE statement shall not reference a unit whose connect team has more than one image.

#### NOTE 9.58

An example of a BACKSPACE statement is:

```
BACKSPACE (10, IOSTAT = N)
```

### 10 9.7.2 ENDFILE statement

- 11 Execution of an ENDFILE statement for a file connected for sequential access writes an endfile record  
12 as the next record of the file. The file is then positioned after the endfile record, which becomes the last  
13 record of the file. If the file also may be connected for direct access, only those records before the endfile  
14 record are considered to have been written. Thus, only those records may be read during subsequent  
15 direct access connections to the file.
- 16 After execution of an ENDFILE statement for a file connected for sequential access, a BACKSPACE  
17 or REWIND statement shall be used to reposition the file prior to execution of any data transfer  
18 input/output statement or ENDFILE statement.
- 19 Execution of an ENDFILE statement for a file connected for stream access causes the terminal point of  
20 the file to become equal to the current file position. Only file storage units before the current position are  
21 considered to have been written; thus only those file storage units may be subsequently read. Subsequent  
22 stream output statements may be used to write further data to the file.
- 23 Execution of an ENDFILE statement for a file that is connected but does not exist creates the file; if  
24 the file is connected for sequential access, it is created prior to writing the endfile record.
- 25 An ENDFILE statement shall not reference a unit whose connect team has more than one image.

#### NOTE 9.59

An example of an ENDFILE statement is:

```
ENDFILE K
```

### 26 9.7.3 REWIND statement

- 27 Execution of a REWIND statement causes the specified file to be positioned at its initial point.



**NOTE 9.60**

If the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

- 1 Execution of a REWIND statement for a file that is connected but does not exist is permitted and has
- 2 no effect on any file.
- 3 A REWIND statement shall not reference a unit whose connect team has more than one image.

**NOTE 9.61**

An example of a REWIND statement is:

```
REWIND 10
```

## 4 9.8 FLUSH statement

- 5 The form of the FLUSH statement is:

6	R927	<i>flush-stmt</i>	<b>is</b>	FLUSH <i>file-unit-number</i>
7			<b>or</b>	FLUSH ( <i>flush-spec-list</i> )
8	R928	<i>flush-spec</i>	<b>is</b>	[UNIT =] <i>file-unit-number</i>
9			<b>or</b>	IOSTAT = <i>scalar-int-variable</i>
10			<b>or</b>	IOMSG = <i>iomsg-variable</i>
11			<b>or</b>	ERR = <i>label</i>

- 12 C945 (R928) No specifier shall appear more than once in a given *flush-spec-list*.

- 13 C946 (R928) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted from
- 14 the unit specifier, the *file-unit-number* shall be the first item in the *flush-spec-list*.

- 15 C947 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement
- 16 that appears in the same scoping unit as the FLUSH statement.

- 17 The IOSTAT=, IOMSG= and ERR= specifiers are described in 9.10. The IOSTAT= variable shall
- 18 be set to a processor-dependent positive value if an error occurs, to zero if the processor-dependent
- 19 flush operation was successful, or to a processor-dependent negative value if the flush operation is not
- 20 supported for the unit specified.

- 21 Execution of a FLUSH statement causes data written to an external unit to be made available to other
- 22 images of the unit's connect team which execute a FLUSH statement in a subsequent segment for that
- 23 unit. It also causes data written to an external file to be available to other processes, or causes data
- 24 placed in an external file by means other than Fortran to be available to a READ statement. The action
- 25 is processor dependent.

- 26 Execution of a FLUSH statement for a file that is connected but does not exist is permitted and has no
- 27 effect on any file. A FLUSH statement has no effect on file position.

**NOTE 9.62**

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 9.63**

An example of a FLUSH statement is:

```
FLUSH( 10, IOSTAT=N)
```

## 9.9 File inquiry

The INQUIRE statement may be used to inquire about properties of a particular named file or of the connection to a particular unit. There are three forms of the INQUIRE statement: **inquire by file**, which uses the FILE= specifier, **inquire by unit**, which uses the UNIT= specifier, and **inquire by output list**, which uses only the IOLENGTH= specifier. All specifier value assignments are performed according to the rules for assignment statements.

An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values assigned by an INQUIRE statement are those that are current at the time the statement is executed.

```
R929  inquire-stmt           is  INQUIRE ( inquire-spec-list )
10                                     or  INQUIRE ( IOLENGTH = scalar-int-variable ) ■
11                                     ■  output-item-list
```

**NOTE 9.64**

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)
```

### 9.9.1 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  inquire-spec           is  [ UNIT = ] file-unit-number
16                                     or  FILE = file-name-expr
17                                     or  ACCESS = scalar-default-char-variable
18                                     or  ACTION = scalar-default-char-variable
19                                     or  ASYNCHRONOUS = scalar-default-char-variable
20                                     or  BLANK = scalar-default-char-variable
21                                     or  DECIMAL = scalar-default-char-variable
22                                     or  DELIM = scalar-default-char-variable
23                                     or  DIRECT = scalar-default-char-variable
24                                     or  ENCODING = scalar-default-char-variable
25                                     or  ERR = label
26                                     or  EXIST = scalar-default-logical-variable
27                                     or  FORM = scalar-default-char-variable
28                                     or  FORMATTED = scalar-default-char-variable
29                                     or  ID = scalar-int-expr
30                                     or  IOMSG = iomsg-variable
31                                     or  IOSTAT = scalar-int-variable
32                                     or  NAME = scalar-default-char-variable
33                                     or  NAMED = scalar-default-logical-variable
34                                     or  NEXTREC = scalar-int-variable
```

- 1                   or NUMBER = *scalar-int-variable*  
 2                   or OPENED = *scalar-default-logical-variable*  
 3                   or PAD = *scalar-default-char-variable*  
 4                   or PENDING = *scalar-default-logical-variable*  
 5                   or POS = *scalar-int-variable*  
 6                   or POSITION = *scalar-default-char-variable*  
 7                   or READ = *scalar-default-char-variable*  
 8                   or READWRITE = *scalar-default-char-variable*  
 9                   or RECL = *scalar-int-variable*  
 10                  or ROUND = *scalar-default-char-variable*  
 11                  or SEQUENTIAL = *scalar-default-char-variable*  
 12                  or SIGN = *scalar-default-char-variable*  
 13                  or SIZE = *scalar-int-variable*  
 14                  or STREAM = *scalar-default-char-variable*  
 15                  or UNFORMATTED = *scalar-default-char-variable*  
 16                  or WRITE = *scalar-default-char-variable*
- 17 C948 (R930) No specifier shall appear more than once in a given *inquire-spec-list*.
- 18 C949 (R930) An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not  
 19 both.
- 20 C950 (R930) In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT=  
 21 are omitted, the *file-unit-number* shall be the first item in the *inquire-spec-list*.
- 22 C951 (R930) If an ID= specifier appears, a PENDING= specifier shall also appear.
- 23 C952 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement  
 24 that appears in the same scoping unit as the INQUIRE statement.
- 25 If *file-unit-number* identifies an internal unit (9.5.3.7.2), an error condition occurs.
- 26 When a returned value of a specifier other than the NAME= specifier is of type character, the value  
 27 returned is in upper case.
- 28 If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier  
 29 variables become undefined, except for variables in the IOSTAT= and IOMSG= specifiers (if any).
- 30 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.
- 31 **9.9.1.1 FILE= specifier in the INQUIRE statement**
- 32 The value of the *file-name-expr* in the FILE= specifier specifies the name of the file being inquired about.  
 33 The named file need not exist or be connected to a unit. The value of the *file-name-expr* shall be of a  
 34 form acceptable to the processor as a file name. Any trailing blanks are ignored. The interpretation of  
 35 case is processor dependent.
- 36 **9.9.1.2 ACCESS= specifier in the INQUIRE statement**
- 37 The *scalar-default-char-variable* in the ACCESS= specifier is assigned the value SEQUENTIAL if the  
 38 file is connected for sequential access, DIRECT if the file is connected for direct access, or STREAM if  
 39 the file is connected for stream access. If there is no connection, it is assigned the value UNDEFINED.
- 40 **9.9.1.3 ACTION= specifier in the INQUIRE statement**
- 41 The *scalar-default-char-variable* in the ACTION= specifier is assigned the value READ if the file is  
 42 connected for input only, WRITE if the file is connected for output only, and READWRITE if it

1 is connected for both input and output. If there is no connection, the *scalar-default-char-variable* is  
2 assigned the value UNDEFINED.

#### 3 **9.9.1.4 ASYNCHRONOUS= specifier in the INQUIRE statement**

4 The *scalar-default-char-variable* in the ASYNCHRONOUS= specifier is assigned the value YES if the  
5 file is connected and asynchronous input/output on the unit is allowed; it is assigned the value NO if the  
6 file is connected and asynchronous input/output on the unit is not allowed. If there is no connection,  
7 the *scalar-default-char-variable* is assigned the value UNDEFINED.

#### 8 **9.9.1.5 BLANK= specifier in the INQUIRE statement**

9 The *scalar-default-char-variable* in the BLANK= specifier is assigned the value ZERO or NULL, corre-  
10 sponding to the blank interpretation mode in effect for a connection for formatted input/output. If there  
11 is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable*  
12 is assigned the value UNDEFINED.

#### 13 **9.9.1.6 DECIMAL= specifier in the INQUIRE statement**

14 The *scalar-default-char-variable* in the DECIMAL= specifier is assigned the value COMMA or POINT,  
15 corresponding to the decimal edit mode in effect for a connection for formatted input/output. If there  
16 is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable*  
17 is assigned the value UNDEFINED.

#### 18 **9.9.1.7 DELIM= specifier in the INQUIRE statement**

19 The *scalar-default-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE, QUOTE,  
20 or NONE, corresponding to the delimiter mode in effect for a connection for formatted input/output.  
21 If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-*  
22 *variable* is assigned the value UNDEFINED.

#### 23 **9.9.1.8 DIRECT= specifier in the INQUIRE statement**

24 The *scalar-default-char-variable* in the DIRECT= specifier is assigned the value YES if DIRECT is  
25 included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of  
26 allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or  
27 not DIRECT is included in the set of allowed access methods for the file.

#### 28 **9.9.1.9 ENCODING= specifier in the INQUIRE statement**

29 The *scalar-default-char-variable* in the ENCODING= specifier is assigned the value UTF-8 if the file  
30 is connected for formatted input/output with an encoding form of UTF-8, and is assigned the value  
31 UNDEFINED if the file is connected for unformatted input/output. If there is no connection, it is  
32 assigned the value UTF-8 if the processor is able to determine that the encoding form of the file is  
33 UTF-8. If the processor is unable to determine the encoding form of the file, the variable is assigned the  
34 value UNKNOWN.

##### **NOTE 9.65**

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).

#### 35 **9.9.1.10 EXIST= specifier in the INQUIRE statement**

36 Execution of an INQUIRE by file statement causes the *scalar-default-logical-variable* in the EXIST=  
37 specifier to be assigned the value true if there exists a file with the specified name; otherwise, false is

1 assigned. Execution of an INQUIRE by unit statement causes true to be assigned if the specified unit  
2 exists; otherwise, false is assigned.

### 3 **9.9.1.11 FORM= specifier in the INQUIRE statement**

4 The *scalar-default-char-variable* in the FORM= specifier is assigned the value FORMATTED if the  
5 file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is  
6 connected for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

### 7 **9.9.1.12 FORMATTED= specifier in the INQUIRE statement**

8 The *scalar-default-char-variable* in the FORMATTED= specifier is assigned the value YES if FORMAT-  
9 TED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set  
10 of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not  
11 FORMATTED is included in the set of allowed forms for the file.

### 12 **9.9.1.13 ID= specifier in the INQUIRE statement**

13 The value of the expression specified in the ID= specifier shall be the identifier of a pending data transfer  
14 operation for the specified unit. This specifier interacts with the PENDING= specifier (9.9.1.20).

### 15 **9.9.1.14 NAME= specifier in the INQUIRE statement**

16 The *scalar-default-char-variable* in the NAME= specifier is assigned the value of the name of the file if  
17 the file has a name; otherwise, it becomes undefined.

#### **NOTE 9.66**

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.

18 The case of the characters assigned to *scalar-default-char-variable* is processor dependent.

### 19 **9.9.1.15 NAMED= specifier in the INQUIRE statement**

20 The *scalar-default-logical-variable* in the NAMED= specifier is assigned the value true if the file has a  
21 name; otherwise, it is assigned the value false.

### 22 **9.9.1.16 NEXTREC= specifier in the INQUIRE statement**

23 The *scalar-int-variable* in the NEXTREC= specifier is assigned the value  $n + 1$ , where  $n$  is the record  
24 number of the last record read from or written to the file connected for direct access. If the file is con-  
25 nected but no records have been read or written since the connection, the *scalar-int-variable* is assigned  
26 the value 1. If the file is not connected for direct access or if the position of the file is indeterminate  
27 because of a previous error condition, the *scalar-int-variable* becomes undefined. If there are pending  
28 data transfer operations for the specified unit, the value assigned is computed as if all the pending data  
29 transfers had already completed.

#### **J3 internal note**

##### **Unresolved Technical Issue 44**

NEXTREC= does not have a well-defined meaning in the context of a unit opened with the TEAM= specifier.

#### 1 9.9.1.17 NUMBER= specifier in the INQUIRE statement

2 The *scalar-int-variable* in the NUMBER= specifier is assigned the value of the external unit number of  
3 the unit that is connected to the file. If there is no unit connected to the file, the value -1 is assigned.

#### 4 9.9.1.18 OPENED= specifier in the INQUIRE statement

5 Execution of an INQUIRE by file statement causes the *scalar-default-logical-variable* in the OPENED=  
6 specifier to be assigned the value true if the file specified is connected to a unit; otherwise, false is  
7 assigned. Execution of an INQUIRE by unit statement causes the *scalar-default-logical-variable* to be  
8 assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

#### 9 9.9.1.19 PAD= specifier in the INQUIRE statement

10 The *scalar-default-char-variable* in the PAD= specifier is assigned the value YES or NO, corresponding  
11 to the pad mode in effect for a connection for formatted input/output. If there is no connection or if  
12 the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value  
13 UNDEFINED.

#### 14 9.9.1.20 PENDING= specifier in the INQUIRE statement

15 The PENDING= specifier is used to determine whether or not previously pending asynchronous data  
16 transfers are complete. A data transfer operation is previously pending if it is pending at the beginning  
17 of execution of the INQUIRE statement.

18 If an ID= specifier appears and the specified data transfer operation is complete, then the variable  
19 specified in the PENDING= specifier is assigned the value false and the INQUIRE statement performs  
20 the wait operation for the specified data transfer.

21 If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit  
22 are complete, then the variable specified in the PENDING= specifier is assigned the value false and the  
23 INQUIRE statement performs wait operations for all previously pending data transfers for the specified  
24 unit.

25 In all other cases, the variable specified in the PENDING= specifier is assigned the value true and no  
26 wait operations are performed; in this case the previously pending data transfers remain pending after  
27 the execution of the INQUIRE statement.

#### NOTE 9.67

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.

#### 28 9.9.1.21 POS= specifier in the INQUIRE statement

29 The *scalar-int-variable* in the POS= specifier is assigned the number of the file storage unit immediately  
30 following the current position of a file connected for stream access. If the file is positioned at its terminal  
31 position, the variable is assigned a value one greater than the number of the highest-numbered file storage

- 1 unit in the file. If the file is not connected for stream access or if the position of the file is indeterminate  
2 because of previous error conditions, the variable becomes undefined.

3 **9.9.1.22 POSITION= specifier in the INQUIRE statement**

4 The *scalar-default-char-variable* in the POSITION= specifier is assigned the value REWIND if the file  
5 is connected by an OPEN statement for positioning at its initial point, APPEND if the file is connected  
6 for positioning before its endfile record or at its terminal point, and ASIS if the file is connected without  
7 changing its position. If there is no connection or if the file is connected for direct access, the *scalar-*  
8 *default-char-variable* is assigned the value UNDEFINED. If the file has been repositioned since the  
9 connection, the *scalar-default-char-variable* is assigned a processor-dependent value, which shall not be  
10 REWIND unless the file is positioned at its initial point and shall not be APPEND unless the file is  
11 positioned so that its endfile record is the next record or at its terminal point if it has no endfile record.

12 **9.9.1.23 READ= specifier in the INQUIRE statement**

13 The *scalar-default-char-variable* in the READ= specifier is assigned the value YES if READ is included  
14 in the set of allowed actions for the file, NO if READ is not included in the set of allowed actions for  
15 the file, and UNKNOWN if the processor is unable to determine whether or not READ is included in  
16 the set of allowed actions for the file.

17 **9.9.1.24 READWRITE= specifier in the INQUIRE statement**

18 The *scalar-default-char-variable* in the READWRITE= specifier is assigned the value YES if READ-  
19 WRITE is included in the set of allowed actions for the file, NO if READWRITE is not included in the  
20 set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or  
21 not READWRITE is included in the set of allowed actions for the file.

22 **9.9.1.25 RECL= specifier in the INQUIRE statement**

23 The *scalar-int-variable* in the RECL= specifier is assigned the value of the record length of a file con-  
24 nected for direct access, or the value of the maximum record length for a file connected for sequential  
25 access. If the file is connected for formatted input/output, the length is the number of characters for all  
26 records that contain only characters of type default character. If the file is connected for unformatted  
27 input/output, the length is measured in file storage units. If there is no connection, or if the connection  
28 is for stream access, the *scalar-int-variable* becomes undefined.

29 **9.9.1.26 ROUND= specifier in the INQUIRE statement**

30 The *scalar-default-char-variable* in the ROUND= specifier is assigned the value UP, DOWN, ZERO,  
31 NEAREST, COMPATIBLE, or PROCESSOR\_DEFINED, corresponding to the I/O rounding mode in  
32 effect for a connection for formatted input/output. If there is no connection or if the connection is not  
33 for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED. The  
34 processor shall return the value PROCESSOR\_DEFINED only if the I/O rounding mode currently in  
35 effect behaves differently than the UP, DOWN, ZERO, NEAREST, and COMPATIBLE modes.

36 **9.9.1.27 SEQUENTIAL= specifier in the INQUIRE statement**

37 The *scalar-default-char-variable* in the SEQUENTIAL= specifier is assigned the value YES if SEQUEN-  
38 TIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included  
39 in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine  
40 whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

#### 1 9.9.1.28 SIGN= specifier in the INQUIRE statement

2 The *scalar-default-char-variable* in the SIGN= specifier is assigned the value PLUS, SUPPRESS, or  
3 PROCESSOR\_DEFINED, corresponding to the sign mode in effect for a connection for formatted in-  
4 put/output. If there is no connection, or if the connection is not for formatted input/output, the  
5 *scalar-default-char-variable* is assigned the value UNDEFINED.

#### 6 9.9.1.29 SIZE= specifier in the INQUIRE statement

7 The *scalar-int-variable* in the SIZE= specifier is assigned the size of the file in file storage units. If the  
8 file size cannot be determined, the variable is assigned the value -1.

9 For a file that may be connected for stream access, the file size is the number of the highest-numbered  
10 file storage unit in the file.

11 For a file that may be connected for sequential or direct access, the file size may be different from the  
12 number of storage units implied by the data in the records; the exact relationship is processor-dependent.

#### 13 9.9.1.30 STREAM= specifier in the INQUIRE statement

14 The *scalar-default-char-variable* in the STREAM= specifier is assigned the value YES if STREAM is  
15 included in the set of allowed access methods for the file, NO if STREAM is not included in the set of  
16 allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or  
17 not STREAM is included in the set of allowed access methods for the file.

#### 18 9.9.1.31 UNFORMATTED= specifier in the INQUIRE statement

19 The *scalar-default-char-variable* in the UNFORMATTED= specifier is assigned the value YES if UN-  
20 FORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included  
21 in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether  
22 or not UNFORMATTED is included in the set of allowed forms for the file.

#### 23 9.9.1.32 WRITE= specifier in the INQUIRE statement

24 The *scalar-default-char-variable* in the WRITE= specifier is assigned the value YES if WRITE is included  
25 in the set of allowed actions for the file, NO if WRITE is not included in the set of allowed actions for  
26 the file, and UNKNOWN if the processor is unable to determine whether or not WRITE is included in  
27 the set of allowed actions for the file.

### 28 9.9.2 Restrictions on inquiry specifiers

29 The *inquire-spec-list* in an INQUIRE by file statement shall contain exactly one FILE= specifier and  
30 shall not contain a UNIT= specifier. The *inquire-spec-list* in an INQUIRE by unit statement shall  
31 contain exactly one UNIT= specifier and shall not contain a FILE= specifier. The unit specified need  
32 not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the  
33 connection and about the file connected.

### 34 9.9.3 Inquire by output list

35 The inquire by output list form of the INQUIRE statement has only an IOLENGTH= specifier and an  
36 output list.

37 The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent number of file  
38 storage units that would be required to store the data of the output list in an unformatted file. The  
39 value shall be suitable as a RECL= specifier in an OPEN statement that connects a file for unformatted  
40 direct access when there are input/output statements with the same input/output list.



- 1 The output list in an INQUIRE statement shall not contain any derived-type list items that require  
 2 a user-defined derived-type input/output procedure as described in subclause 9.5.2. If a derived-type  
 3 list item appears in the output list, the value returned for the IOLENGTH= specifier assumes that no  
 4 user-defined derived-type input/output procedure will be invoked.

## 5 **9.10 Error, end-of-record, and end-of-file conditions**

6 The set of input/output error conditions is processor dependent.

7 An **end-of-record condition** occurs when a nonadvancing input statement attempts to transfer data  
 8 from a position beyond the end of the current record, unless the file is a stream file and the current  
 9 record is at the end of the file (an end-of-file condition occurs instead).

10 An **end-of-file condition** occurs when

- 11 (1) an endfile record is encountered during the reading of a file connected for sequential access,
- 12 (2) an attempt is made to read a record beyond the end of an internal file, or
- 13 (3) an attempt is made to read beyond the end of a stream file.

14 An end-of-file condition may occur at the beginning of execution of an input statement. An end-of-file  
 15 condition also may occur during execution of a formatted input statement when more than one record  
 16 is required by the interaction of the input list and the format. An end-of-file condition also may occur  
 17 during execution of a stream input statement.

### 18 **9.10.1 Error conditions and the ERR= specifier**

19 If an error condition occurs during execution of an input/output statement, the position of the file  
 20 becomes indeterminate.

21 If an error condition occurs during execution of an input/output statement that contains neither an  
 22 ERR= nor IOSTAT= specifier, execution of the program is terminated. If an error condition occurs  
 23 during execution of an input/output statement that contains either an ERR= specifier or an IOSTAT=  
 24 specifier then

- 25 (1) processing of the input/output list, if any, terminates,
- 26 (2) if the statement is a data transfer statement or the error occurs during a wait operation, all  
 27 *do-variables* in the statement that initiated the transfer become undefined,
- 28 (3) if an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes  
 29 defined as specified in 9.10.4,
- 30 (4) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
- 31 (5) if the statement is a READ statement and it contains a SIZE= specifier, the *scalar-int-*  
 32 *variable* in the SIZE= specifier becomes defined as specified in 9.5.1.14,
- 33 (6) if the statement is a READ statement or the error condition occurs in a wait operation for  
 34 a transfer initiated by a READ statement, all input items or namelist group objects in the  
 35 statement that initiated the transfer become undefined, and
- 36 (7) if an ERR= specifier appears, execution continues with the statement labeled by the *label*  
 37 in the ERR= specifier.

### 38 **9.10.2 End-of-file condition and the END= specifier**

39 If an end-of-file condition occurs during execution of an input/output statement that contains neither  
 40 an END= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-file  
 41 condition occurs during execution of an input/output statement that contains either an END= specifier  
 42 or an IOSTAT= specifier, and an error condition does not occur then

- 1 (1) processing of the input list, if any, terminates,
- 2 (2) if the statement is a data transfer statement or the error occurs during a wait operation, all
- 3 *do-variables* in the statement that initiated the transfer become undefined,
- 4 (3) if the statement is a READ statement or the end-of-file condition occurs in a wait operation
- 5 for a transfer initiated by a READ statement, all input list items or namelist group objects
- 6 in the statement that initiated the transfer become undefined,
- 7 (4) if the file specified in the input statement is an external record file, it is positioned after the
- 8 endfile record,
- 9 (5) if an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes
- 10 defined as specified in 9.10.4,
- 11 (6) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
- 12 and
- 13 (7) if an END= specifier appears, execution continues with the statement labeled by the *label*
- 14 in the END= specifier.

### 15 9.10.3 End-of-record condition and the EOR= specifier

16 If an end-of-record condition occurs during execution of an input/output statement that contains neither  
 17 an EOR= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-  
 18 record condition occurs during execution of an input/output statement that contains either an EOR=  
 19 specifier or an IOSTAT= specifier, and an error condition does not occur then

- 20 (1) If the pad mode has the value
  - 21 (a) YES, the record is padded with blanks to satisfy the input list item (9.5.3.4.2) and
  - 22 corresponding data edit descriptor that requires more characters than the record con-
  - 23 tains,
  - 24 (b) NO, the input list item becomes undefined,
- 25 (2) processing of the input list, if any, terminates,
- 26 (3) if the statement is a data transfer statement or the error occurs during a wait operation, all
- 27 *do-variables* in the statement that initiated the transfer become undefined,
- 28 (4) the file specified in the input statement is positioned after the current record,
- 29 (5) if an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes
- 30 defined as specified in 9.10.4,
- 31 (6) if an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
- 32 (7) if a SIZE= specifier appears, the *scalar-int-variable* in the SIZE= specifier becomes defined
- 33 as specified in (9.5.1.14), and
- 34 (8) if an EOR= specifier appears, execution continues with the statement labeled by the *label*
- 35 in the EOR= specifier.

### 36 9.10.4 IOSTAT= specifier

37 Execution of an input/output statement containing the IOSTAT= specifier causes the *scalar-int-variable*  
 38 in the IOSTAT= specifier to become defined with

- 39 (1) a zero value if neither an error condition, an end-of-file condition, nor an end-of-record
- 40 condition occurs,
- 41 (2) the processor-dependent positive integer value of the constant IOSTAT\_INQUIRE\_INTERNAL\_-
- 42 UNIT from the ISO\_FORTRAN\_ENV intrinsic module (13.8.3) if a unit number in an IN-
- 43 QUIRE statement identifies an internal file,
- 44 (3) a processor-dependent positive integer value different from IOSTAT\_INQUIRE\_INTERNAL\_-
- 45 UNIT if any other error condition occurs,

- 1 (4) the processor-dependent negative integer value of the constant IOSTAT\_END (13.8.3.10) if
- 2 an end-of-file condition occurs and no error condition occurs, or
- 3 (5) the processor-dependent negative integer value of the constant IOSTAT\_EOR (13.8.3.11) if
- 4 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

```

**5 9.10.5 IOMSG= specifier**

- 6 If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement,
- 7 the processor shall assign an explanatory message to *iomsg-variable*. If no such condition occurs, the
- 8 processor shall not change the value of *iomsg-variable*.

**9 9.11 Restrictions on input/output statements**

- 10 If a unit, or a file connected to a unit, does not have all of the properties required for the execution of
- 11 certain input/output statements, those statements shall not refer to the unit.
- 12 An input/output statement that is executed while another input/output statement is being executed is
- 13 called a **recursive input/output statement**.
- 14 A recursive input/output statement shall not identify an external unit that is identified by another
- 15 input/output statement being executed except that a child data transfer statement may identify its
- 16 parent data transfer statement external unit.
- 17 An input/output statement shall not cause the value of any established format specification to be
- 18 modified.
- 19 A recursive input/output statement shall not modify the value of any internal unit except that a recursive
- 20 WRITE statement may modify the internal unit identified by that recursive WRITE statement.
- 21 The value of a specifier in an input/output statement shall not depend on any *input-item*, *io-implied-*
- 22 *do do-variable*, or on the definition or evaluation of any other specifier in the *io-control-spec-list* or
- 23 *inquire-spec-list* in that statement.
- 24 The value of any subscript or substring bound of a variable that appears in a specifier in an input/output
- 25 statement shall not depend on any *input-item*, *io-implied-do do-variable*, or on the definition or evalua-
- 26 tion of any other specifier in the *io-control-spec-list* or *inquire-spec-list* in that statement.
- 27 In a data transfer statement, the variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier, if
- 28 any, shall not be associated with any entity in the data transfer input/output list (9.5.2) or *namelist-*
- 29 *group-object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.

- 1 In a data transfer statement, if a variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier is an
- 2 array element reference, its subscript values shall not be affected by the data transfer, the *io-implied-do*
- 3 processing, or the definition or evaluation of any other specifier in the *io-control-spec-list*.
- 4 A variable that may become defined or undefined as a result of its use in a specifier in an INQUIRE
- 5 statement, or any associated entity, shall not appear in another specifier in the same INQUIRE statement.
- 6 A STOP statement shall not be executed during execution of an input/output statement.

**NOTE 9.69**

Restrictions on the evaluation of expressions (7.1.8) prohibit certain side effects.
--------------------------------------------------------------------------------------

## 1 10 Input/output editing

### 2 10.1 Format specifications

3 A format used in conjunction with an input/output statement provides information that directs the  
4 editing between the internal representation of data and the characters of a sequence of formatted records.

5 A FMT= specifier (9.5.1.1) in an input/output statement may refer to a FORMAT statement or to a  
6 character expression that contains a format specification. A format specification provides explicit editing  
7 information. The FMT= specifier alternatively may be an asterisk (\*), which indicates list-directed  
8 formatting (10.10). Namelist formatting (10.11) may be indicated by specifying a *namelist-group-name*  
9 instead of a *format*.

### 10 10.2 Explicit format specification methods

#### 11 10.2.1 FORMAT statement

12 R1001 *format-stmt* is FORMAT *format-specification*  
13 R1002 *format-specification* is ( [ *format-item-list* ] )  
14 or ( [ *format-item-list*, ] *unlimited-format-item* )

15 C1001 (R1001) The *format-stmt* shall be labeled.

16 C1002 (R1002) The comma used to separate *format-items* in a *format-item-list* may be omitted

- 17 (1) between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit  
18 descriptor (10.8.5), possibly preceded by a repeat specifier,
- 19 (2) before a slash edit descriptor when the optional repeat specification is not present (10.8.2),
- 20 (3) after a slash edit descriptor, or
- 21 (4) before or after a colon edit descriptor (10.8.3)

22 Blank characters may precede the initial left parenthesis of the format specification. Additional blank  
23 characters may appear at any point within the format specification, with no effect on the interpretation  
24 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))
```

#### 25 10.2.2 Character format specification

26 A character expression used as a *format* in a formatted input/output statement shall evaluate to a  
27 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) *r* shall be positive.

18 C1004 (R1005) *r* shall not have a kind parameter specified for it.

19 The integer literal constant *r* 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**.

- |                                |    |                           |
|--------------------------------|----|---------------------------|
| 23 R1006 <i>data-edit-desc</i> | is | I <i>w</i> [ . <i>m</i> ] |
| 24                             | or | B <i>w</i> [ . <i>m</i> ] |

1                    **or** O *w* [ . *m* ]  
 2                    **or** Z *w* [ . *m* ]  
 3                    **or** F *w* . *d*  
 4                    **or** E *w* . *d* [ E *e* ]  
 5                    **or** EN *w* . *d* [ E *e* ]  
 6                    **or** ES *w* . *d* [ E *e* ]  
 7                    **or** G *w* [ . *d* [ E *e* ] ]  
 8                    **or** L *w*  
 9                    **or** A [ *w* ]  
 10                   **or** D *w* . *d*  
 11                   **or** DT [ *char-literal-constant* ] [ ( *v-list* ) ]  
 12 R1007    *w*                    **is** *int-literal-constant*  
 13 R1008    *m*                    **is** *int-literal-constant*  
 14 R1009    *d*                    **is** *int-literal-constant*  
 15 R1010    *e*                    **is** *int-literal-constant*  
 16 R1011    *v*                    **is** *signed-int-literal-constant*  
  
 17 C1005    (R1010) *e* shall be positive.  
  
 18 C1006    (R1007) *w* shall be zero or positive for the I, B, O, Z, F, and G edit descriptors. *w* shall be  
 19            positive for all other edit descriptors.  
  
 20 C1007    (R1006) For the G edit descriptor, *d* shall be specified if and only if *w* is not zero.  
  
 21 C1008    (R1006) *w*, *m*, *d*, *e*, and *v* shall not have kind parameters specified for them.  
  
 22 C1009    (R1006) The *char-literal-constant* in the DT edit descriptor shall not have a kind parameter  
 23            specified for it.  
  
 24 I, B, O, Z, F, E, EN, ES, G, L, A, D, and DT indicate the manner of editing.  
  
 25 R1012    *control-edit-desc*            **is** *position-edit-desc*  
 26                                            **or** [ *r* ] /  
 27                                            **or** :  
 28                                            **or** *sign-edit-desc*  
 29                                            **or** *k* P  
 30                                            **or** *blank-interp-edit-desc*  
 31                                            **or** *round-edit-desc*  
 32                                            **or** *decimal-edit-desc*  
 33 R1013    *k*                    **is** *signed-int-literal-constant*  
  
 34 C1010    (R1013) *k* shall not have a kind parameter specified for it.  
  
 35 In *k* P, *k* is called the **scale factor**.  
  
 36 R1014    *position-edit-desc*            **is** T *n*  
 37                                            **or** TL *n*  
 38                                            **or** TR *n*  
 39                                            **or** *n* X  
 40 R1015    *n*                    **is** *int-literal-constant*  
  
 41 C1011    (R1015) *n* shall be positive.  
  
 42 C1012    (R1015) *n* shall not have a kind parameter specified for it.  
  
 43 R1016    *sign-edit-desc*            **is** SS  
 44                                            **or** SP

1			<b>or</b>	S
2	R1017	<i>blank-interp-edit-desc</i>	<b>is</b>	BN
3			<b>or</b>	BZ
4	R1018	<i>round-edit-desc</i>	<b>is</b>	RU
5			<b>or</b>	RD
6			<b>or</b>	RZ
7			<b>or</b>	RN
8			<b>or</b>	RC
9			<b>or</b>	RP
10	R1019	<i>decimal-edit-desc</i>	<b>is</b>	DC
11			<b>or</b>	DP

12 T, TL, TR, X, slash, colon, SS, SP, S, P, BN, BZ, RU, RD, RZ, RN, RC, RP, DC, and DP indicate the  
13 manner of editing.

```
14 R1020 char-string-edit-desc is char-literal-constant
```

15 C1013 (R1020) The *char-literal-constant* shall not have a kind parameter specified for it.

16 Each *rep-char* in a character string edit descriptor shall be one of the characters capable of representation  
17 by the processor.

18 The character string edit descriptors provide constant data to be output, and are not valid for input.

19 The edit descriptors are without regard to case except for the characters in the character constants.

20 **10.3.3 Fields**

21 A **field** is a part of a record that is read on input or written on output when format control encounters  
22 a data edit descriptor or a character string edit descriptor. The **field width** is the size in characters of  
23 the field.

## 24 10.4 Interaction between input/output list and format

25 The start of formatted data transfer using a format specification initiates **format control** (9.5.3.4.2).

26 Each action of format control depends on information jointly provided by the next edit descriptor in the  
27 format specification and the next effective item in the input/output list, if one exists.

28 If an input/output list specifies at least one effective list item, at least one data edit descriptor shall  
29 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 (9.5.3.4). Zero length character items are effective list items, but zero sized arrays and implied DO lists with an iteration count of zero are not.

30 A format specification is interpreted from left to right. The exceptions are format items preceded by a  
31 repeat specification  $r$ , and format reversion (described below).

32 A format item preceded by a repeat specification is processed as a list of  $r$  items, each identical to the  
33 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.

- 1 To each data edit descriptor interpreted in a format specification, there corresponds one effective item
- 2 specified by the input/output list (9.5.2), except that an input/output list item of type complex requires
- 3 the interpretation of two F, E, EN, ES, D, or G edit descriptors. For each control edit descriptor or
- 4 character edit descriptor, there is no corresponding item specified by the input/output list, and format
- 5 control communicates information directly with the record.
  
- 6 Whenever format control encounters a data edit descriptor in a format specification, it determines
- 7 whether there is a corresponding effective item specified by the input/output list. If there is such an
- 8 item, it transmits appropriately edited information between the item and the record, and then format
- 9 control proceeds. If there is no such item, format control terminates.
  
- 10 If format control encounters a colon edit descriptor in a format specification and another effective in-
- 11 put/output list item is not specified, format control terminates.
  
- 12 If format control encounters the rightmost parenthesis of a complete format specification and another
- 13 effective input/output list item is not specified, format control terminates. However, if another effective
- 14 input/output list item is specified, format control then reverts to the beginning of the format item
- 15 terminated by the last preceding right parenthesis that is not part of a DT edit descriptor. If there
- 16 is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format
- 17 specification. If any reversion occurs, the reused portion of the format specification shall contain at
- 18 least one data edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat
- 19 specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on
- 20 the changeable modes (9.4.1). If format control reverts to a parenthesis that is not the beginning of an
- 21 *unlimited-format-item*, the file is positioned in a manner identical to the way it is positioned when a
- 22 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.

For example,

```
WRITE( 10, '( "IARRAY =", *( I0, :, ",") )' ) IARRAY
```

produces a single record with a header and a comma separated list of integer values.

## 1 10.5 Positioning by format control

2 After each data edit descriptor or character string edit descriptor is processed, the file is positioned after  
3 the last character read or written in the current record.

4 After each T, TL, TR, or X edit descriptor is processed, the file is positioned as described in 10.8.1.

5 After each slash edit descriptor is processed, the file is positioned as described in 10.8.2.

6 During formatted stream output, processing of an A edit descriptor may cause file positioning to occur  
7 (10.7.4).

8 If format control reverts as described in 10.4, the file is positioned in a manner identical to the way it is  
9 positioned when a slash edit descriptor is processed (10.8.2).

10 During a read operation, any unprocessed characters of the current record are skipped whenever the  
11 next record is read.

## 12 10.6 Decimal symbol

13 The **decimal symbol** is the character that separates the whole and fractional parts in the decimal  
14 representation of a real number in an internal or external file. When the decimal edit mode is POINT,  
15 the decimal symbol is a decimal point. When the decimal edit mode is COMMMA, the decimal symbol is  
16 a comma.

## 17 10.7 Data edit descriptors

### 18 10.7.1 General

19 Data edit descriptors cause the conversion of data to or from its internal representation; during formatted  
20 stream output, the A data edit descriptor may also cause file positioning. On input, the specified variable  
21 becomes defined unless an error condition, an end-of-file condition, or an end-of-record condition occurs.  
22 On output, the specified expression is evaluated.

23 During input from a Unicode file,

- 24 (1) characters in the record that correspond to an ASCII character variable shall have a position  
25 in the ISO 10646 character type collating sequence of 127 or less, and
- 26 (2) characters in the record that correspond to a default character variable shall be representable  
27 in the default character type.

28 During input from a non-Unicode file,

- 29 (1) characters in the record that correspond to a character variable shall have the kind of the  
30 character variable, and
- 31 (2) characters in the record that correspond to a numeric logical, or bits variable shall be of  
32 default character type.

33 During output to a Unicode file, all characters transmitted to the record are of ISO 10646 character  
34 type. If a character input/output list item or character string edit descriptor contains a character that  
35 is not representable in the ISO 10646 character type, the result is processor-dependent.

36 During output to a non-Unicode file, characters transmitted to the record as a result of processing a  
37 character string edit descriptor or as a result of evaluating a numeric, logical, bits, or default character  
38 data entity, are of type default character.

## 1 10.7.2 Numeric and bits editing

### 2 10.7.2.1 General rules

3 The I, B, O, Z, F, E, EN, ES, D, and G edit descriptors may be used to specify the input/output of  
4 integer, real, complex, and bits data. The following general rules apply.

- 5 (1) On input, leading blanks are not significant. When the input field is not an IEEE excep-  
6 tional specification (10.7.2.3.2), the interpretation of blanks, other than leading blanks, is  
7 determined by the blank interpretation mode (10.8.6). Plus signs may be omitted. A field  
8 containing only blanks is considered to be zero.
- 9 (2) On input, with F, E, EN, ES, D, and G editing, a decimal symbol appearing in the input  
10 field overrides the portion of an edit descriptor that specifies the decimal symbol location.  
11 The input field may have more digits than the processor uses to approximate the value of  
12 the datum.
- 13 (3) On output with I, F, E, EN, ES, D, and G editing, the representation of a positive or zero  
14 internal value in the field may be prefixed with a plus sign, as controlled by the S, SP, and  
15 SS edit descriptors or the processor. The representation of a negative internal value in the  
16 field shall be prefixed with a minus sign.
- 17 (4) On output, the representation is right justified in the field. If the number of characters  
18 produced by the editing is smaller than the field width, leading blanks are inserted in the  
19 field.
- 20 (5) On output, if the number of characters produced exceeds the field width or if an exponent  
21 exceeds its specified length using the *Ew.d Ee*, *ENw.d Ee*, *ESw.d Ee*, or *Gw.d Ee* edit  
22 descriptor, the processor shall fill the entire field of width *w* with asterisks. However,  
23 the processor shall not produce asterisks if the field width is not exceeded when optional  
24 characters are omitted.

#### NOTE 10.8

When an SP edit descriptor is in effect, a plus sign is not optional.
-----------------------------------------------------------------------

- 25 (6) On output, with I, B, O, Z, F, and G editing, the specified value of the field width *w* may  
26 be zero. In such cases, the processor selects the smallest positive actual field width that  
27 does not result in a field filled with asterisks. The specified value of *w* shall not be zero on  
28 input.

### 29 10.7.2.2 Integer editing

30 The *Iw* and *Iw.m,* edit descriptors indicate that the field to be edited occupies *w* positions, except when  
31 *w* is zero. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero. The  
32 specified input/output list item shall be of type integer or bits. The G, B, O, and Z edit descriptor also  
33 may be used to edit integer data (10.7.5.2.1, 10.7.2.4).

34 If the input list item is of type bits, the input field is edited as if the input list item were of type integer  
35 and bits compatible with the actual list item, followed by the effect of an intrinsic assignment of the  
36 resulting integer value to the actual list item.

**J3 internal note (cont.)****J3 internal note**

## Unresolved Technical Issue 45

This is misplaced and/or misguided. The type and kind of the input item do not affect input editing (read through the existing subclause - nowhere does this even mention the type or more pertinantly kind of the input item).

Nor does input editing define the list item, that is "data transfer" (04-007: 9.5.3.4 and in particular 9.5.3.4.2).

Worse, one of the bits paper's own edits makes a change to how input editing is done (!) depending on the type of the list item, but the "as if" rule above means that is ineffective.

**J3 internal note**

## Unresolved Technical Issue 46

This proposal appears to be requiring unsigned integers by the back door. However, since we don't actually have unsigned integers it does not work. Consider, for example, an 8-bit bits variable, reading the value "130". According to the above, this is equivalent to reading it into a one-byte integer - but that is not valid, 130 not being in the set of values. So it may fail.

**J3 internal note**

## Unresolved Technical Issue 47

This feature is unusable by the normal user. If he has a bits variable of `KIND==7`, what is the effect? It is all very well saying "as if" there was an integer variable of the right kind, but that kind is not guaranteed to exist and might well not exist.

- 1 On input, *m* has no effect.
- 2 In the input field for the *I* edit descriptor, the character string shall be a *signed-digit-string* (R409) if
- 3 the list item is of type integer and a *digit-string* (R410) if it is of type bits, except for the interpretation
- 4 of blanks.
- 5 The output field for the *Iw* edit descriptor consists of zero or more leading blanks followed by a minus
- 6 sign if the internal value is negative, or an optional plus sign otherwise, followed by the magnitude of
- 7 the internal value as a *digit-string* without leading zeros.

**NOTE 10.9**

A *digit-string* always consists of at least one digit.

- 8 The output field for the *Iw.m* edit descriptor is the same as for the *Iw* edit descriptor, except that the
- 9 *digit-string* consists of at least *m* digits. If necessary, sufficient leading zeros are included to achieve the
- 10 minimum of *m* digits. The value of *m* shall not exceed the value of *w*, except when *w* is zero. If *m* is
- 11 zero and the internal value is zero, the output field consists of only blank characters, regardless of the
- 12 sign control in effect. When *m* and *w* are both zero, and the internal value is zero, one blank character
- 13 is produced.

**14 10.7.2.3 Real and complex editing****15 10.7.2.3.1 General**

- 16 The F, E, EN, ES, and D edit descriptors specify the editing of real and complex data. An input/output
- 17 list item corresponding to an F, E, EN, ES, or D edit descriptor shall be real or complex. The G, B, O,
- 18 and Z edit descriptors also may be used to edit real and complex data (10.7.5.2.2).

- 19 A lower-case letter is equivalent to the corresponding upper-case letter in an IEEE exceptional specifi-

1 cation or the exponent in a numeric input field.

## 2 10.7.2.3.2 F editing

3 The *Fw.d* edit descriptor indicates that the field occupies *w* positions, the fractional part of which  
4 consists of *d* digits. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero.

5 The input field is either an IEEE exceptional specification or consists of an optional sign, followed by a  
6 string of one or more digits optionally containing a decimal symbol, including any blanks interpreted as  
7 zeros. The *d* has no effect on input if the input field contains a decimal symbol. If the decimal symbol  
8 is omitted, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted  
9 as the fractional part of the value represented. The string of digits may contain more digits than a  
10 processor uses to approximate the value. The basic form may be followed by an exponent of one of the  
11 following forms:

- 12 (1) a *sign* followed by a *digit-string*;
- 13 (2) the letter E followed by zero or more blanks, followed by a *signed-digit-string*;
- 14 (3) the letter D followed by zero or more blanks, followed by a *signed-digit-string*.

15 An exponent containing a D is processed identically to an exponent containing an E.

### NOTE 10.10

If the input field does not contain an exponent, the effect is as if the basic form were followed by an exponent with a value of  $-k$ , where *k* is the established scale factor (10.8.5).

16 An input field that is an IEEE exceptional specification consists of optional blanks, followed by either

- 17 (1) an optional sign, followed by the string 'INF' or the string 'INFINITY', or
- 18 (2) an optional sign, followed by the string 'NaN', optionally followed by zero or more alphanu-  
19 meric characters enclosed in parentheses,

20 optionally followed by blanks.

21 The value specified by form (1) is an IEEE infinity; this form shall not be used if the processor does  
22 not support IEEE infinities for the input variable. The value specified by form (2) is an IEEE NaN;  
23 this form shall not be used if the processor does not support IEEE NaNs for the input variable. The  
24 NaN value is a quiet NaN if the only nonblank characters in the field are 'NaN' or 'NaN()'; otherwise,  
25 the NaN value is processor-dependent. The interpretation of a sign in a NaN input field is processor  
26 dependent.

27 For an internal value that is an IEEE infinity, the output field consists of blanks, if necessary, followed  
28 by a minus sign for negative infinity or an optional plus sign otherwise, followed by the letters 'Inf' or  
29 'Infinity', right justified within the field. If *w* is less than 3, the field is filled with asterisks; otherwise,  
30 if *w* is less than 8, 'Inf' is produced.

31 For an internal value that is an IEEE NaN, the output field consists of blanks, if necessary, followed by  
32 the letters 'NaN' and optionally followed by one to  $w - 5$  alphanumeric processor-dependent characters  
33 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.

34 For an internal value that is neither an IEEE infinity nor a NaN, the output field consists of blanks, if  
35 necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise,

1 followed by a string of digits that contains a decimal symbol and represents the magnitude of the internal  
 2 value, as modified by the established scale factor and rounded to  $d$  fractional digits. Leading zeros are  
 3 not permitted except for an optional zero immediately to the left of the decimal symbol if the magnitude  
 4 of the value in the output field is less than one. The optional zero shall appear if there would otherwise  
 5 be no digits in the output field.

### 6 10.7.2.3.3 E and D editing

7 The  $Ew.d$ ,  $Dw.d$ , and  $Ew.d Ee$  edit descriptors indicate that the external field occupies  $w$  positions, the  
 8 fractional part of which consists of  $d$  digits, unless a scale factor greater than one is in effect, and the  
 9 exponent part consists of  $e$  digits. The  $e$  has no effect on input.

10 The form and interpretation of the input field is the same as for  $Fw.d$  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  $Fw.d$ .

13 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field for a scale  
 14 factor of zero is:

15  $[ \pm ] [0].x_1x_2 \dots x_d exp$

16 where:

17  $\pm$  signifies a plus sign or a minus sign.

18  $.$  signifies a decimal symbol (10.6).

19  $x_1x_2 \dots x_d$  are the  $d$  most significant digits of the internal value after rounding.

20  $exp$  is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
$Ew.d$	$ 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 $\pm 0z_1z_2$
	$99 <  exp  \leq 999$	$\pm z_1z_2z_3$

21 where each  $z$  is a digit.

22 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit  
 23 descriptor forms  $Ew.d$  and  $Dw.d$  shall not be used if  $|exp| > 999$ .

24 The scale factor  $k$  controls the decimal normalization (10.3.2, 10.8.5). If  $-d < k \leq 0$ , the output field  
 25 contains exactly  $|k|$  leading zeros and  $d - |k|$  significant digits after the decimal symbol. If  $0 < k < d + 2$ ,  
 26 the output field contains exactly  $k$  significant digits to the left of the decimal symbol and  $d - k + 1$   
 27 significant digits to the right of the decimal symbol. Other values of  $k$  are not permitted.

### 28 10.7.2.3.4 EN editing

29 The EN edit descriptor produces an output field in the form of a real number in engineering notation  
 30 such that the decimal exponent is divisible by three and the absolute value of the significand (R414) is  
 31 greater than or equal to 1 and less than 1000, except when the output value is zero. The scale factor  
 32 has no effect on output.

33 The forms of the edit descriptor are  $ENw.d$  and  $ENw.d Ee$  indicating that the external field occupies  $w$

1 positions, the fractional part of which consists of  $d$  digits and the exponent part consists of  $e$  digits.

2 The form and interpretation of the input field is the same as for *Fw.d* editing (10.7.2.3.2).

3 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for  
4 *Fw.d*.

5 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is:

6  $[ \pm ] \, yyy \, . \, x_1 x_2 \dots x_d \, exp$

7 where:

8  $\pm$  signifies a plus sign or a minus sign.

9  $yyy$  are the 1 to 3 decimal digits representative of the most significant digits of the internal  
10 value after rounding ( $yyy$  is an integer such that  $1 \leq yyy < 1000$  or, if the output value is  
11 zero,  $yyy = 0$ ).

12  $.$  signifies a decimal symbol (10.6).

13  $x_1 x_2 \dots x_d$  are the  $d$  next most significant digits of the internal value after rounding.

14  $exp$  is a decimal exponent, divisible by three, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
EN <i>w.d</i>	$ exp  \leq 99$	$E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 <  exp  \leq 999$	$\pm z_1 z_2 z_3$
EN <i>w.d Ee</i>	$ exp  \leq 10^e - 1$	$E\pm z_1 z_2 \dots z_e$

15 where each  $z$  is a digit.

16 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit  
17 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

#### 18 10.7.2.3.5 ES editing

19 The ES edit descriptor produces an output field in the form of a real number in scientific notation such  
20 that the absolute value of the significand (R414) is greater than or equal to 1 and less than 10, except  
21 when the output value is zero. The scale factor has no effect on output.

22 The forms of the edit descriptor are ES*w.d* and ES*w.d Ee* indicating that the external field occupies  $w$   
23 positions, the fractional part of which consists of  $d$  digits and the exponent part consists of  $e$  digits.

24 The form and interpretation of the input field is the same as for *Fw.d* editing (10.7.2.3.2).

25 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for  
26 *Fw.d*.

27 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is:

1  $[ \pm ] y . x_1 x_2 \dots x_d \exp$

2 where:

3  $\pm$  signifies a plus sign or a minus sign.

4  $y$  is a decimal digit representative of the most significant digit of the internal value after  
5 rounding.

6  $.$  signifies a decimal symbol (10.6).

7  $x_1 x_2 \dots x_d$  are the  $d$  next most significant digits of the internal value after rounding.

8  $\exp$  is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
$ESw.d$	$ exp  \leq 99$	$E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 <  exp  \leq 999$	$\pm z_1 z_2 z_3$
$ESw.d Ee$	$ exp  \leq 10^e - 1$	$E\pm z_1 z_2 \dots z_e$

9 where each  $z$  is a digit.

10 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit  
11 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

#### 12 10.7.2.3.6 Complex editing

13 A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex type  
14 is specified by two edit descriptors each of which specifies the editing of real data. The first of the edit  
15 descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may  
16 be different. Control and character string edit descriptors may be processed between the edit descriptor  
17 for the real part and the edit descriptor for the imaginary part.

#### 18 10.7.2.3.7 Rounding mode

19 The rounding mode can be specified by an OPEN statement (9.4.1), a data transfer input/output  
20 statement (9.5.1.12), or an edit descriptor (10.8.7).

21 In what follows, the term “decimal value” means the exact decimal number as given by the character  
22 string, while the term “internal value” means the number actually stored (typically in binary form) in  
23 the processor. For example, in dealing with the decimal constant 0.1, the decimal value is the exact  
24 mathematical quantity  $1/10$ , which has no exact representation on most processors. Formatted output  
25 of real data involves conversion from an internal value to a decimal value; formatted input involves  
26 conversion from a decimal value to an internal value.

27 When the I/O rounding mode is UP, the value resulting from conversion shall be the smallest repre-  
28 sentable value that is greater than or equal to the original value. When the I/O rounding mode is  
29 DOWN, the value resulting from conversion shall be the largest representable value that is less than or



equal to the original value. When the I/O rounding mode is ZERO, the value resulting from conversion shall be the value closest to the original value and no greater in magnitude than the original value. When the I/O rounding mode is NEAREST, the value resulting from conversion shall be the closer of the two nearest representable values if one is closer than the other. If the two nearest representable values are equidistant from the original value, it is processor dependent which one of them is chosen. When the I/O rounding mode is COMPATIBLE, the value resulting from conversion shall be the closer of the two nearest representable values or the value away from zero if halfway between them. When the I/O rounding mode is PROCESSOR\_DEFINED, rounding during conversion shall be a processor dependent default mode, which may correspond to one of the other modes.

On processors that support IEEE rounding on conversions, NEAREST shall correspond to round to 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.

#### 10.7.2.4 Bits editing

The *Bw*, *Bw.m*, *Ow*, *Ow.m*, *Zw*, and *Zw.m* edit descriptors indicate that the field to be edited occupies *w* positions, except when *w* is zero. When *w* is zero, the processor selects the field width. On input, *w* shall not be zero. The input/output list item shall be of type bits, integer, real, complex, or logical. 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.

If the input list item is not of type bits, the input field is edited as if the input list item were of type bits and bits compatible with the actual list item.

On input, *m* has no effect.

In the input field for the B, O, and Z edit descriptors the character string shall consist of binary, octal, or hexadecimal digits (as in R426, R427, R428) in the respective input field. The lower-case hexadecimal digits a through f in a hexadecimal input field are equivalent to the corresponding upper-case hexadecimal digits.

If the output list item is not of type bits, it is first converted to a list item of type bits that is bits compatible with it.

If the output list item, *x*, is not of type bits, it is interpreted as if it were of type bits with the value BITS(*x*).

The output field for the *Bw*, *Ow*, and *Zw* descriptors consists of zero or more leading blanks followed by the internal value in a form identical to the digits of a binary, octal, or hexadecimal constant, respectively, 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.

R1021 *hex-digit-string* is *hex-digit* [ *hex-digit* ] ...

The output field for the *Bw.m*, *Ow.m*, and *Zw.m* edit descriptor is the same as for the *Bw*, *Ow*, and *Zw* edit descriptor, except that the *digit-string* or *hex-digit-string* consists of at least *m* digits. If necessary,

- 1 sufficient leading zeros are included to achieve the minimum of  $m$  digits. The value of  $m$  shall not exceed  
 2 the value of  $w$ , except when  $w$  is zero. If  $m$  is zero and the internal value consists of all zero bits, the  
 3 output field consists of only blank characters. When  $m$  and  $w$  are both zero, and the internal value  
 4 consists of all zero bits, one blank character is produced.

### 5 10.7.3 Logical editing

- 6 The Lw edit descriptor indicates that the field occupies  $w$  positions. The specified input/output list  
 7 item shall be of type logical. The G edit descriptor also may be used to edit logical data (10.7.5.4).  
 8 The input field consists of optional blanks, optionally followed by a period, followed by a T for true or  
 9 F for false. The T or F may be followed by additional characters in the field, which are ignored.  
 10 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.

- 11 The output field consists of  $w - 1$  blanks followed by a T or F, depending on whether the internal value  
 12 is true or false, respectively.

### 13 10.7.4 Character editing

- 14 The A[w] edit descriptor is used with an input/output list item of type character. The G edit descriptor  
 15 also may be used to edit character data (10.7.5.5). The kind type parameter of all characters transferred  
 16 and converted under control of one A or G edit descriptor is implied by the kind of the corresponding  
 17 list item.  
 18 If a field width  $w$  is specified with the A edit descriptor, the field consists of  $w$  characters. If a field  
 19 width  $w$  is not specified with the A edit descriptor, the number of characters in the field is the length of  
 20 the corresponding list item, regardless of the value of the kind type parameter.  
 21 Let  $len$  be the length of the input/output list item. If the specified field width  $w$  for an A edit descriptor  
 22 corresponding to an input item is greater than or equal to  $len$ , the rightmost  $len$  characters will be taken  
 23 from the input field. If the specified field width  $w$  is less than  $len$ , the  $w$  characters will appear left  
 24 justified with  $len - w$  trailing blanks in the internal value.  
 25 If the specified field width  $w$  for an A edit descriptor corresponding to an output item is greater than  
 26  $len$ , the output field will consist of  $w - len$  blanks followed by the  $len$  characters from the internal value.  
 27 If the specified field width  $w$  is less than or equal to  $len$ , the output field will consist of the leftmost  $w$   
 28 characters from the internal value.

#### NOTE 10.17

For nondefault character types, the blank padding character is processor dependent.

- 29 If the file is connected for stream access, the output may be split across more than one record if it  
 30 contains newline characters. A newline character is a nonblank character returned by the intrinsic  
 31 function NEW\_LINE. Beginning with the first character of the output field, each character that is not  
 32 a newline is written to the current record in successive positions; each newline character causes file  
 33 positioning at that point as if by slash editing (the current record is terminated at that point, a new  
 34 empty record is created following the current record, this new record becomes the last and current record  
 35 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.

**1 10.7.5 Generalized editing****2 10.7.5.1 Overview**

3 The *Gw*, *Gw.d* and *Gw.d Ee* edit descriptors are used with an input/output list item of any intrinsic  
 4 type. When *w* is nonzero, these edit descriptors indicate that the external field occupies *w* positions,  
 5 the fractional part of which consists of a maximum of *d* digits and the exponent part consists of *e* digits.  
 6 When these edit descriptors are used to specify the input/output of integer, logical, bits, or character  
 7 data, *d* and *e* have no effect. When *w* is zero the processor selects the field width. On input, *w* shall  
 8 not be zero.”

**9 10.7.5.2 Generalized numeric editing**

10 When used to specify the input/output of integer, real, and complex data, the *Gw*, *Gw.d* and *Gw.d Ee*  
 11 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.

**12 10.7.5.2.1 Generalized integer editing**

13 When used to specify the input/output of integer data, the *Gw.d* and *Gw.d Ee* edit descriptors follow  
 14 the rules for the *Iw* edit descriptor (10.7.2.2), except that *w* shall not be zero. When used to specify the  
 15 output of integer data, the *G0* edit descriptor follows the rules for the *I0* edit descriptor.

**16 10.7.5.2.2 Generalized real and complex editing**

17 The form and interpretation of the input field is the same as for *Fw.d* editing (10.7.2.3.2).

18 When used to specify the output of real or complex data, the *G0* edit descriptor follows the rules for  
 19 the *ESw.d Ee* edit descriptor. Reasonable processor-dependent values of *w*, *d*, and *e* are used with each  
 20 output value.

21 For an internal value that is an IEEE infinity or NaN, the form of the output field for the *Gw.d* and  
 22 *Gw.d Ee* edit descriptors is the same as for *Fw.d*.

23 Otherwise, the method of representation in the output field depends on the magnitude of the internal  
 24 value being edited. Let *N* be the magnitude of the internal value and *r* be the rounding mode value  
 25 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  
 26 *d* is 0, *Gw.d* output editing is the same as *k PEw.d* output editing and *Gw.d Ee* output editing is  
 27 the same as *k PEw.d Ee* output editing, where *k* is the scale factor (10.8.5) currently in effect. If  
 28  $0.1 - r \times 10^{-d-1} \leq N < 10^d - r$  or *N* is identically 0 and *d* is not zero, the scale factor has no effect,  
 29 and the value of *N* determines the editing as follows:

Magnitude of Internal Value	Equivalent Conversion
$N = 0$	$F(w - n).(d - 1), n('b')$
$0.1 - r \times 10^{-d-1} \leq N < 1 - r \times 10^{-d}$	$F(w - n).d, n('b')$
$1 - r \times 10^{-d} \leq N < 10 - r \times 10^{-d+1}$	$F(w - n).(d - 1), n('b')$
$10 - r \times 10^{-d+1} \leq N < 100 - r \times 10^{-d+2}$	$F(w - n).(d - 2), n('b')$

Magnitude of Internal Value	Equivalent Conversion
.	.
.	.
.	.
$10^{d-2} - r \times 10^{-2} \leq N < 10^{d-1} - r \times 10^{-1}$	F( $w - n$ ).1, $n('b')$ )
$10^{d-1} - r \times 10^{-1} \leq N < 10^d - r$	F( $w - n$ ).0, $n('b')$ )

- 1 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  
2 follows:

I/O Rounding Mode	$r$
COMPATIBLE	0.5
NEAREST	0.5 if the higher value is even −0.5 if the lower value is even
UP	1
DOWN	0
ZERO	1 if internal value is negative 0 if internal value is positive

- 3 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.

#### 4 10.7.5.3 Generalized bits editing

- 5 When used to specify the input/output of bits data, the  $Gw.d$  and  $Gw.d Ee$  edit descriptors follow the  
6 rules for the  $Zw$  edit descriptor (10.7.2.4), except that  $w$  shall not be zero. When used to specify the  
7 output of bits data, the  $G0$  edit descriptor follows the rules for the  $Z0$  edit descriptor.

#### 8 10.7.5.4 Generalized logical editing

- 9 When used to specify the input/output of logical data, the  $Gw.d$  and  $Gw.d Ee$  edit descriptors follow  
10 the rules for the  $Lw$  edit descriptor (10.7.3). When used to specify the output of logical data, the  $G0$   
11 edit descriptor follows the rules for the  $L1$  edit descriptor.

#### 12 10.7.5.5 Generalized character editing

- 13 When used to specify the input/output of character data, the  $Gw.d$  and  $Gw.d Ee$  edit descriptors follow  
14 the rules for the  $Aw$  edit descriptor (10.7.4). When used to specify the output of character data, the  $G0$   
15 edit descriptor follows the rules for the  $A$  edit descriptor with no field width.

#### 16 10.7.6 User-defined derived-type editing

- 17 The DT edit descriptor allows a user-provided procedure to be used instead of the processor's default  
18 input/output formatting for processing a list item of derived type.

- 19 The DT edit descriptor may include a character literal constant. The character value "DT" concatenated  
20 with the character literal constant is passed to the user-defined derived-type input/output procedure as  
21 the `iotype` argument (9.5.3.7). The  $v$  values of the edit descriptor are passed to the user-defined  
22 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/)`.

- 1 If a derived-type variable or value corresponds to a DT edit descriptor, there shall be an accessible  
 2 interface to a corresponding derived-type input/output procedure for that derived type (9.5.3.7). A DT  
 3 edit descriptor shall not correspond with a list item that is not of a derived type.

## 4 **10.8 Control edit descriptors**

- 5 A control edit descriptor does not cause the transfer of data or the conversion of data to or from internal  
 6 representation, but may affect the conversions performed by subsequent data edit descriptors.

### 7 **10.8.1 Position editing**

- 8 The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted  
 9 to or from the record. If any character skipped by a T, TL, TR, or X edit descriptor is of type nondefault  
 10 character, and the unit is an internal file of type default character or an external non-Unicode file, the  
 11 result of that position editing is processor dependent.
- 12 The position specified by a T edit descriptor may be in either direction from the current position. On  
 13 input, this allows portions of a record to be processed more than once, possibly with different editing.
- 14 The position specified by an X edit descriptor is forward from the current position. On input, a position  
 15 beyond the last character of the record may be specified if no characters are transmitted from such  
 16 positions.

**NOTE 10.22**

An `nX` edit descriptor has the same effect as a `TRn` edit descriptor.

- 17 On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and  
 18 therefore does not by itself affect the length of the record. If characters are transmitted to positions at  
 19 or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously  
 20 filled are filled with blanks. The result is as if the entire record were initially filled with blanks.
- 21 On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never  
 22 directly causes a character already placed in the record to be replaced. Such edit descriptors may result  
 23 in positioning such that subsequent editing causes a replacement.

#### 24 **10.8.1.1 T, TL, and TR editing**

- 25 The **left tab limit** affects file positioning by the T and TL edit descriptors. Immediately prior to  
 26 nonchild data transfer, the left tab limit becomes defined as the character position of the current record  
 27 or the current position of the stream file. If, during data transfer, the file is positioned to another record,  
 28 the left tab limit becomes defined as character position one of that record.
- 29 The `Tn` edit descriptor indicates that the transmission of the next character to or from a record is to  
 30 occur at the *n*th character position of the record, relative to the left tab limit.
- 31 The `TLn` edit descriptor indicates that the transmission of the next character to or from the record is  
 32 to occur at the character position *n* characters backward from the current position. However, if *n* is  
 33 greater than the difference between the current position and the left tab limit, the `TLn` edit descriptor  
 34 indicates that the transmission of the next character to or from the record is to occur at the left tab  
 35 limit.

- 1 The  $TR_n$  edit descriptor indicates that the transmission of the next character to or from the record is  
 2 to occur at the character position  $n$  characters forward from the current position.

**NOTE 10.23**

The  $n$  in a  $T_n$ ,  $TL_n$ , or  $TR_n$  edit descriptor shall be specified and shall be greater than zero.

### 3 **10.8.1.2 X editing**

- 4 The  $nX$  edit descriptor indicates that the transmission of the next character to or from a record is to  
 5 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.

## 6 **10.8.2 Slash editing**

- 7 The slash edit descriptor indicates the end of data transfer to or from the current record.
- 8 On input from a file connected for sequential or stream access, the remaining portion of the current  
 9 record is skipped and the file is positioned at the beginning of the next record. This record becomes  
 10 the current record. On output to a file connected for sequential or stream access, a new empty record  
 11 is created following the current record; this new record then becomes the last and current record of the  
 12 file and the file is positioned at the beginning of this new record.
- 13 For a file connected for direct access, the record number is increased by one and the file is positioned  
 14 at the beginning of the record that has that record number, if there is such a record, and this record  
 15 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.

- 16 The repeat specification is optional in the slash edit descriptor. If it is not specified, the default value is  
 17 one.

## 18 **10.8.3 Colon editing**

- 19 The colon edit descriptor terminates format control if there are no more effective items in the in-  
 20 put/output list (9.5.2). The colon edit descriptor has no effect if there are more effective items in the  
 21 input/output list.

## 22 **10.8.4 SS, SP, and S editing**

- 23 The SS, SP, and S edit descriptors temporarily change (9.4.1) the sign mode (9.4.6.5, 9.5.1.13) for the  
 24 connection. The edit descriptors SS, SP, and S set the sign mode corresponding to the SIGN= specifier  
 25 values SUPPRESS, PLUS, and PROCESSOR\_DEFINED, respectively.

- 26 The sign mode controls optional plus characters in numeric output fields. When the sign mode is PLUS,  
 27 the processor shall produce a plus sign in any position that normally contains an optional plus sign.  
 28 When the sign mode is SUPPRESS, the processor shall not produce a plus sign in such positions. When  
 29 the sign mode is PROCESSOR\_DEFINED, the processor has the option of producing a plus sign or not  
 30 in such positions, subject to 10.7.2(5).

1 The SS, SP, and S edit descriptors affect only I, F, E, EN, ES, D, and G editing during the execution of  
 2 an output statement. The SS, SP, and S edit descriptors have no effect during the execution of an input  
 3 statement.

#### 4 **10.8.5 P editing**

5 The  $k$ P edit descriptor temporarily changes (9.4.1) the scale factor for the connection to  $k$ . The scale  
 6 factor affects the editing of F, E, EN, ES, D, and G edit descriptors for numeric quantities.

7 The scale factor  $k$  affects the appropriate editing in the following manner.

- 8 (1) On input, with F, E, EN, ES, D, and G editing (provided that no exponent exists in the  
 9 field) and F output editing, the scale factor effect is that the externally represented number  
 10 equals the internally represented number multiplied by  $10^k$ .
- 11 (2) On input, with F, E, EN, ES, D, and G editing, the scale factor has no effect if there is an  
 12 exponent in the field.
- 13 (3) On output, with E and D editing, the significand (R414) part of the quantity to be produced  
 14 is multiplied by  $10^k$  and the exponent is reduced by  $k$ .
- 15 (4) On output, with G editing, the effect of the scale factor is suspended unless the magnitude  
 16 of the datum to be edited is outside the range that permits the use of F editing. If the use  
 17 of E editing is required, the scale factor has the same effect as with E output editing.
- 18 (5) On output, with EN and ES editing, the scale factor has no effect.

19 If UP, DOWN, ZERO, or NEAREST I/O rounding mode is in effect,

- 20 (1) on input, the scale factor is applied to the external decimal value and then this is converted  
 21 using the current I/O rounding mode, and
- 22 (2) on output, the internal value is converted using the current I/O rounding mode and then  
 23 the scale factor is applied to the converted decimal value.

#### 24 **10.8.6 BN and BZ editing**

25 The BN and BZ edit descriptors temporarily change (9.4.1) the blank interpretation mode (9.4.5.4,  
 26 9.5.1.5) for the connection. The edit descriptors BN and BZ set the blank interpretation mode corre-  
 27 sponding to the BLANK= specifier values NULL and ZERO, respectively.

28 The blank interpretation mode controls the interpretation of nonleading blanks in numeric and bits  
 29 input fields. Such blank characters are interpreted as zeros when the blank interpretation mode has the  
 30 value ZERO; they are ignored when the blank interpretation mode has the value NULL. The effect of  
 31 ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the  
 32 field right justified, and the blanks replaced as leading blanks. However, a field containing only blanks  
 33 has the value zero.

34 The blank interpretation mode affects only numeric editing, bits editing, generalized numeric editing,  
 35 and generalized bits editing on input. It has no effect on output.

#### 36 **10.8.7 RU, RD, RZ, RN, RC, and RP editing**

37 The round edit descriptors temporarily change (9.4.1) the connection's I/O rounding mode (9.4.6.4,  
 38 9.5.1.12, 10.7.2.3.7). The round edit descriptors RU, RD, RZ, RN, RC, and RP set the I/O rounding  
 39 mode corresponding to the ROUND= specifier values UP, DOWN, ZERO, NEAREST, COMPATIBLE,  
 40 and PROCESSOR\_DEFINED, respectively. The I/O rounding mode affects the conversion of real and  
 41 complex values in formatted input/output. It affects only D, E, EN, ES, F, and G editing.

### 1 10.8.8 DC and DP editing

2 The decimal edit descriptors temporarily change (9.4.1) the decimal edit mode (9.4.5.5, 9.5.1.6) for the  
3 connection. The edit descriptors DC and DP set the decimal edit mode corresponding to the DECIMAL=  
4 specifier values COMMA and POINT, respectively.

5 The decimal edit mode controls the representation of the decimal symbol (10.6) during conversion of  
6 real and complex values in formatted input/output. The decimal edit mode affects only D, E, EN, ES,  
7 F, and G editing. If the mode is COMMA during list-directed input/output, the character used as a  
8 value separator is a semicolon in place of a comma.

## 9 10.9 Character string edit descriptors

10 A character string edit descriptor shall not be used on input.

11 The character string edit descriptor causes characters to be written from the enclosed characters of the  
12 edit descriptor itself, including blanks. For a character string edit descriptor, the width of the field is  
13 the number of characters between the delimiting characters. Within the field, two consecutive delimiting  
14 characters are counted as a single character.

#### NOTE 10.26

A delimiter for a character string edit descriptor is either an apostrophe or quote.

## 15 10.10 List-directed formatting

16 List-directed input/output allows data editing according to the type of the list item instead of by a  
17 format specification. It also allows data to be free-field, that is, separated by commas (or semicolons)  
18 or blanks.

19 The characters in one or more list-directed records constitute a sequence of values and value separators.  
20 The end of a record has the same effect as a blank character, unless it is within a character constant. Any  
21 sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character  
22 constant.

23 Each value is either a null value,  $c$ ,  $r^*c$ , or  $r^*$ , where  $c$  is a literal constant, optionally signed if integer  
24 or real, or an undelimited character constant and  $r$  is an unsigned, nonzero, integer literal constant.  
25 Neither  $c$  nor  $r$  shall have kind type parameters specified. The constant  $c$  is interpreted as though  
26 it had the same kind type parameter as the corresponding list item. The  $r^*c$  form is equivalent to  $r$   
27 successive appearances of the constant  $c$ , and the  $r^*$  form is equivalent to  $r$  successive appearances of  
28 the null value. Neither of these forms may contain embedded blanks, except where permitted within the  
29 constant  $c$ .

30 A **value separator** is

- 31 (1) a comma optionally preceded by one or more contiguous blanks and optionally followed by  
32 one or more contiguous blanks, unless the decimal edit mode is COMMA, in which case a  
33 semicolon is used in place of the comma,
- 34 (2) a slash optionally preceded by one or more contiguous blanks and optionally followed by  
35 one or more contiguous blanks, or
- 36 (3) one or more contiguous blanks between two nonblank values or following the last nonblank  
37 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 **10.10.1 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 unlimited 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  
 18 by an ordered pair of numeric input fields separated by a separator, and followed by a right parenthesis.  
 19 The first numeric input field is the real part of the complex constant and the second is the imaginary  
 20 part. Each of the numeric input fields may be preceded or followed by any number of blanks and ends of  
 21 records. The separator is a comma if the decimal edit mode is POINT; it is a semicolon if the decimal  
 22 edit mode is COMMA. The end of a record may occur between the real part and the separator or between  
 23 the separator and the imaginary part.

24 When the next effective item is of type logical, the input form shall not include value separators among  
 25 the optional characters permitted for L editing.

26 When the next effective item is of type character, the input form consists of a possibly delimited sequence  
 27 of zero or more *rep-chars* whose kind type parameter is implied by the kind of the effective list item.  
 28 Character sequences may be continued from the end of one record to the beginning of the next record,  
 29 but the end of record shall not occur between a doubled apostrophe in an apostrophe-delimited character  
 30 sequence, nor between a doubled quote in a quote-delimited character sequence. The end of the record  
 31 does not cause a blank or any other character to become part of the character sequence. The character  
 32 sequence may be continued on as many records as needed. The characters blank, comma, semicolon,  
 33 and slash may appear in default, ASCII, or ISO 10646 character sequences.

- 1 If the next effective item is of type default, ASCII, or ISO 10646 character and
- 2 (1) the character sequence does not contain value separators,
  - 3 (2) the character sequence does not cross a record boundary,
  - 4 (3) the first nonblank character is not a quotation mark or an apostrophe,
  - 5 (4) the leading characters are not *digits* followed by an asterisk, and
  - 6 (5) the character sequence contains at least one character,
- 7 the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the  
 8 character sequence is terminated by the first blank, comma (if the decimal edit mode is POINT),  
 9 semicolon (if the decimal edit mode is COMMA), slash, or end of record; in this case apostrophes and  
 10 quotation marks within the datum are not to be doubled.

11 Let *len* be the length of the next effective item, and let *w* be the length of the character sequence. If  
 12 *len* is less than or equal to *w*, the leftmost *len* characters of the sequence are transmitted to the next  
 13 effective item. If *len* is greater than *w*, the sequence is transmitted to the leftmost *w* characters of the  
 14 next effective item and the remaining *len*−*w* characters of the next effective item are filled with blanks.  
 15 The effect is as though the sequence were assigned to the next effective item in a character assignment  
 16 statement (7.4.1.3).

#### 17 10.10.1.1 Null values

18 A null value is specified by

- 19 (1) the *r*\* form,
- 20 (2) no characters between consecutive value separators, or
- 21 (3) no characters before the first value separator in the first record read by each execution of a  
 22 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.

23 A null value has no effect on the definition status of the next effective item. A null value shall not be  
 24 used for either the real or imaginary part of a complex constant, but a single null value may represent  
 25 an entire complex constant.

26 A slash encountered as a value separator during execution of a list-directed input statement causes  
 27 termination of execution of that input statement after the assignment of the previous value. Any  
 28 characters remaining in the current record are ignored. If there are additional items in the input list,  
 29 the effect is as if null values had been supplied for them. Any *do-variable* in the input list is defined as  
 30 though 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$
```

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

### 1 10.10.2 List-directed output

- 2 The form of the values produced is the same as that required for input, except as noted otherwise. With
- 3 the exception of adjacent undelimited character sequences, the values are separated by one or more
- 4 blanks or by a comma, or a semicolon if the decimal edit mode is comma, optionally preceded by one or
- 5 more blanks and optionally followed by one or more blanks.
- 6 The processor may begin new records as necessary, but the end of record shall not occur within a
- 7 constant except for complex constants and character sequences. The processor shall not insert blanks
- 8 within character sequences or within constants, except for complex constants.
- 9 Logical output values are T for the value true and F for the value false.
- 10 Integer output constants are produced with the effect of an *Iw* edit descriptor.
- 11 Bits output constants are produced with the effect of a *Zw* edit descriptor with *w* equal to  $\frac{k+3}{4}$  where
- 12 *k* is the kind type parameter value of the list item.

**J3 internal note (cont.)****J3 internal note**

## Unresolved Technical Issue 48

The above formula was based on the idea that leading zeros of bit values are significant. However, this cannot be true, since they are suppressed using the normal edit descriptors and in particular for B0 O0 and Z0.

Therefore I recommend changing the above to Z0.

Actually, there is another question, which is why is it that bits values are apparently “naturally” written with Z notation? I would have expected B notation.

Furthermore, the above formula is ambiguous. Does it mean  $\lceil \frac{k+3}{4} \rceil$ ,  $\lfloor \frac{k+3}{4} \rfloor$ , or  $\lceil \frac{k+3}{4} \rceil$ ? Should we stick to something like this formula, I think it should be  $\lceil \frac{k}{4} \rceil$ .

BTW, when this gets changed, the corresponding change should be made for namelist output.

- 1 Real constants are produced with the effect of either an F edit descriptor or an E edit descriptor,
- 2 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-
- 3 dependent integers. If the magnitude  $x$  is within this range or is zero, the constant is produced using
- 4  $0PFw.d$ ; otherwise,  $1PEw.d Ee$  is used.
  
- 5 For numeric output, reasonable processor-dependent values of  $w$ ,  $d$ , and  $e$  are used for each of the
- 6 numeric constants output.
  
- 7 Complex constants are enclosed in parentheses with a separator between the real and imaginary parts,
- 8 each produced as defined above for real constants. The separator is a comma if the decimal edit mode is
- 9 POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between
- 10 the separator and the imaginary part only if the entire constant is as long as, or longer than, an entire
- 11 record. The only embedded blanks permitted within a complex constant are between the separator and
- 12 the end of a record and one blank at the beginning of the next record.
  
- 13 Character sequences produced when the delimiter mode has a value of NONE
  
- 14     (1) are not delimited by apostrophes or quotation marks,
- 15     (2) are not separated from each other by value separators,
- 16     (3) have each internal apostrophe or quotation mark represented externally by one apostrophe
- 17         or quotation mark, and
- 18     (4) have a blank character inserted by the processor at the beginning of any record that begins
- 19         with the continuation of a character sequence from the preceding record.
  
- 20 Character sequences produced when the delimiter mode has a value of QUOTE are delimited by quotes,
- 21 are preceded and followed by a value separator, and have each internal quote represented on the external
- 22 medium by two contiguous quotes.
  
- 23 Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited
- 24 by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe
- 25 represented on the external medium by two contiguous apostrophes.
  
- 26 If two or more successive values in an output record have identical values, the processor has the option
- 27 of producing a repeated constant of the form  $r*c$  instead of the sequence of identical values.
  
- 28 Slashes, as value separators, and null values are not produced as output by list-directed formatting.
  
- 29 Except for continuation of delimited character sequences, each output record begins with a blank char-
- 30 acter.

**NOTE 10.33**

The length of the output records is not specified exactly and may be processor dependent.

## 10.11 Namelist formatting

Namelist input/output allows data editing with NAME=value subsequences. This facilitates documentation of input and output files and more flexibility on input.

The characters in one or more namelist records constitute a sequence of **name-value subsequences**, each of which consists of an object designator followed by an equals and followed by one or more values and value separators. The equals may optionally be preceded or followed by one or more contiguous blanks. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

The name may be any name in the *namelist-group-object-list* (5.6).

Each value is either a null value (10.11.1.4),  $c$ ,  $r^*c$ , or  $r^*$ , where  $c$  is a literal constant, optionally signed if integer or real, and  $r$  is an unsigned, nonzero, integer literal constant. A kind type parameter shall not be specified for  $c$  or  $r$ . The constant  $c$  is interpreted as though it had the same kind type parameter as the corresponding 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 null values. Neither of these forms may contain embedded blanks, except where permitted within the constant  $c$ .

A value separator for namelist formatting is the same as for list-directed formatting (10.10).

### 10.11.1 Namelist input

Input for a namelist input statement consists of

- (1) optional blanks and namelist comments,
- (2) the character & followed immediately by the *namelist-group-name* as specified in the NAMELIST statement,
- (3) one or more blanks,
- (4) a sequence of zero or more name-value subsequences separated by value separators, and
- (5) a slash to terminate the namelist input.

**NOTE 10.34**

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.

In each name-value subsequence, the name shall be the name of a namelist group object list item with an optional qualification and the name with the optional qualification shall not be a zero-sized array, a zero-sized array section, or a zero-length character string. The optional qualification, if any, shall not contain a vector subscript.

A group name or object name is without regard to case.

#### 10.11.1.1 Namelist group object names

Within the input data, each name shall correspond to a particular namelist group object name. Subscripts, strides, and substring range expressions used to qualify group object names shall be optionally signed integer literal constants with no kind type parameters specified. If a namelist group object is

1 an array, the input record corresponding to it may contain either the array name or the designator of  
2 a subobject of that array, using the syntax of object designators (R603). If the namelist group object  
3 name is the name of a variable of derived type, the name in the input record may be either the name of  
4 the variable or the designator of one of its components, indicated by qualifying the variable name with  
5 the appropriate component name. Successive qualifications may be applied as appropriate to the shape  
6 and type of the variable represented.

7 The order of names in the input records need not match the order of the namelist group object items.  
8 The input records need not contain all the names of the namelist group object items. The definition  
9 status of any names from the *namelist-group-object-list* that do not occur in the input record remains  
10 unchanged. In the input record, each object name or subobject designator may be preceded and followed  
11 by one or more optional blanks but shall not contain embedded blanks.

#### 12 10.11.1.2 Namelist input values

13 The datum *c* (10.11) is any input value acceptable to format specifications for a given type, except for  
14 a restriction on the form of input values corresponding to list items of types logical, integer, bits, and  
15 character as specified in 10.11.1.3. The form of a real or complex value is dependent on the decimal edit  
16 mode in effect (10.8.8). The form of an input value shall be acceptable for the type of the namelist group  
17 object list item. The number and forms of the input values that may follow the equals in a name-value  
18 subsequence depend on the shape and type of the object represented by the name in the input record.  
19 When the name in the input record is that of a scalar variable of an intrinsic type, the equals shall  
20 not be followed by more than one value. Blanks are never used as zeros, and embedded blanks are not  
21 permitted in constants except within character constants and complex constants as specified in 10.11.1.3.

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  
26 that formatted input/output list items are expanded (9.5.2). Each input value following the equals shall  
27 then be acceptable to format specifications for the type of the list item in the corresponding position in  
28 the expanded sequence, except as noted in 10.11.1.3. 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 assignment 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 10.11.1.3 Namelist group object list items

2 When the next effective namelist group object list item is of type real, the input form of the input value  
3 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  
4 assumed to have no fractional digits unless a decimal symbol appears within the field.

5 When the next effective item is of type complex, the input form of the input value consists of a left  
6 parenthesis followed by an ordered pair of numeric input fields separated by a comma (if the decimal  
7 edit mode is POINT) or a semicolon (if the decimal edit mode is COMMA), and followed by a right  
8 parenthesis. The first numeric input field is the real part of the complex constant and the second part  
9 is the imaginary part. Each of the numeric input fields may be preceded or followed by any number of  
10 blanks and ends of records. The end of a record may occur between the real part and the comma or  
11 semicolon, or between the comma or semicolon and the imaginary part.

12 When the next effective item is of type logical, the input form of the input value shall not include equals  
13 or value separators among the optional characters permitted for L editing (10.7.3).

14 When the next effective item is of type integer, the value in the input record is interpreted as if an *Iw*  
15 edit descriptor with a suitable value of *w* were used.

16 When the next effective list item is of type bits, the value in the input record is interpreted as if a *Zw*  
17 edit descriptor with a suitable value of *w* were used.

18 When the next effective item is of type character, the input form consists of a delimited sequence of zero  
19 or more *rep-chars* whose kind type parameter is implied by the kind of the corresponding list item. Such  
20 a sequence may be continued from the end of one record to the beginning of the next record, but the  
21 end of record shall not occur between a doubled apostrophe in an apostrophe-delimited sequence, nor  
22 between a doubled quote in a quote-delimited sequence. The end of the record does not cause a blank or  
23 any other character to become part of the sequence. The sequence may be continued on as many records  
24 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).

25 Let *len* be the length of the next effective item, and let *w* be the length of the character sequence. If  
26 *len* is less than or equal to *w*, the leftmost *len* characters of the sequence are transmitted to the next  
27 effective item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the  
28 next effective item and the remaining *len-w* characters of the next effective item are filled with blanks.  
29 The effect is as though the sequence were assigned to the next effective item in a character assignment  
30 statement (7.4.1.3).

### 31 10.11.1.4 Null values

32 A null value is specified by

- 33 (1) the *r\** form,
- 34 (2) blanks between two consecutive value separators following an equals,
- 35 (3) zero or more blanks preceding the first value separator and following an equals, or
- 36 (4) two consecutive nonblank value separators.

37 A null value has no effect on the definition status of the corresponding input list item. If the namelist  
38 group object list item is defined, it retains its previous value; if it is undefined, it remains undefined. A

- 1 null value shall not be used as either the real or imaginary part of a complex constant, but a single null  
 2 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.

**3 10.11.1.5 Blanks**

- 4 All blanks in a namelist input record are considered to be part of some value separator except for
- 5 (1) blanks embedded in a character constant,
  - 6 (2) embedded blanks surrounding the real or imaginary part of a complex constant,
  - 7 (3) leading blanks following the equals unless followed immediately by a slash or comma, or a  
 8 semicolon if the decimal edit mode is comma, and
  - 9 (4) blanks between a name and the following equals.

**10 10.11.1.6 Namelist Comments**

- 11 Except within a character literal constant, a “!” character after a value separator or in the first nonblank  
 12 position of a namelist input record initiates a comment. The comment extends to the end of the current  
 13 input record and may contain any graphic character in the processor-dependent character set. The  
 14 comment is ignored. A slash within the namelist comment does not terminate execution of the namelist  
 15 input statement. Namelist comments are not allowed in stream input because comments depend on  
 16 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:

```
&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 10.11.2 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 10.11.2.1 Namelist output editing

12 Logical output values are T for the value true and F for the value false.

13 Integer output constants are produced with the effect of an *Iw* edit descriptor.

14 Bits output constants are produced with the effect of a *Zw* edit descriptor with *w* equal to  $\frac{k+3}{4}$  where  
15 *k* is the kind type parameter value of the list item.

16 Real constants are produced with the effect of either an *F* edit descriptor or an *E* edit descriptor,  
17 depending on the magnitude *x* of the value and a range  $10^{d_1} \leq x < 10^{d_2}$ , where *d*<sub>1</sub> and *d*<sub>2</sub> are processor-  
18 dependent integers. If the magnitude *x* is within this range or is zero, the constant is produced using  
19 *0PFw.d*; otherwise, *1PEw.d Ee* is used.

20 For numeric output, reasonable processor-dependent integer values of *w*, *d*, and *e* are used for each of  
21 the numeric constants output.

22 Complex constants are enclosed in parentheses with a separator between the real and imaginary parts,  
23 each produced as defined above for real constants. The separator is a comma if the decimal edit mode is  
24 POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between  
25 the separator and the imaginary part only if the entire constant is as long as, or longer than, an entire  
26 record. The only embedded blanks permitted within a complex constant are between the separator and  
27 the end of a record and one blank at the beginning of the next record.

28 Character sequences produced when the delimiter mode has a value of NONE

- 29 (1) are not delimited by apostrophes or quotation marks,
- 30 (2) are not separated from each other by value separators,
- 31 (3) have each internal apostrophe or quotation mark represented externally by one apostrophe  
32 or quotation mark, and
- 33 (4) have a blank character inserted by the processor at the beginning of any record that begins  
34 with the continuation of a character sequence from the preceding record.

### 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.

35 Character sequences produced when the delimiter mode has a value of QUOTE are delimited by quotes,

1 are preceded and followed by a value separator, and have each internal quote represented on the external  
2 medium by two contiguous quotes.

3 Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited  
4 by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe  
5 represented on the external medium by two contiguous apostrophes.

#### 6 **10.11.2.2 Namelist output records**

7 If two or more successive values in an array in an output record produced have identical values, the  
8 processor has the option of producing a repeated constant of the form  $r*c$  instead of the sequence of  
9 identical values.

10 The name of each namelist group object list item is placed in the output record followed by an equals  
11 and a list of values of the namelist group object list item.

12 An ampersand character followed immediately by a *namelist-group-name* will be produced by namelist  
13 formatting at the start of the first output record to indicate which particular group of data objects is  
14 being output. A slash is produced by namelist formatting to indicate the end of the namelist formatting.

15 A null value is not produced by namelist formatting.

16 Except for new records created by explicit formatting within a user-defined derived-type output pro-  
17 cedure or by continuation of delimited character sequences, each output record begins with a blank  
18 character.

# 11 Program units

The terms and basic concepts of program units were introduced in 2.2. A program unit is a main program, an external subprogram, a module, a submodule, or a block data program unit.

This clause describes main programs, modules, submodules, and block data program units. Clause 12 describes external subprograms.

## 11.1 Main program

A Fortran **main program** is a program unit that does not contain a SUBROUTINE, FUNCTION, MODULE, 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*.

C1103 (R1101) An automatic object shall not appear in the *specification-part* (R204) of a main program.

### 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

1 itself.

## 2 11.2 Modules

3 A **module** contains specifications and definitions that are to be accessible to other program units. A  
 4 module that is provided as an inherent part of the processor is an **intrinsic module**. A **nonintrinsic**  
 5 **module** is defined by a module program unit or a means other than Fortran.

6 Procedures and types defined in an intrinsic module are not themselves intrinsic.

- |    |       |                               |           |                                       |
|----|-------|-------------------------------|-----------|---------------------------------------|
| 7  | R1104 | <i>module</i>                 | <b>is</b> | <i>module-stmt</i>                    |
| 8  |       |                               |           | [ <i>specification-part</i> ]         |
| 9  |       |                               |           | [ <i>module-subprogram-part</i> ]     |
| 10 |       |                               |           | <i>end-module-stmt</i>                |
| 11 | R1105 | <i>module-stmt</i>            | <b>is</b> | MODULE <i>module-name</i>             |
| 12 | R1106 | <i>end-module-stmt</i>        | <b>is</b> | END [ MODULE [ <i>module-name</i> ] ] |
| 13 | R1107 | <i>module-subprogram-part</i> | <b>is</b> | <i>contains-stmt</i>                  |
| 14 |       |                               |           | [ <i>module-subprogram</i> ] ...      |
| 15 | R1108 | <i>module-subprogram</i>      | <b>is</b> | <i>function-subprogram</i>            |
| 16 |       |                               | <b>or</b> | <i>subroutine-subprogram</i>          |
| 17 |       |                               | <b>or</b> | <i>separate-module-subprogram</i>     |
- 18 C1104 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the  
 19 *module-name* specified in the *module-stmt*.
- 20 C1105 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a  
 21 *format-stmt*.
- 22 C1106 (R1104) An automatic object shall not appear in the *specification-part* of a module.
- 23 C1107 (R1104) If an object of a type for which *component-initialization* is specified (R448) appears  
 24 in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER  
 25 attribute, the object shall have the SAVE attribute.

### 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.

A module is host to any module subprograms (12.2.2.2) it contains, and the entities in the module are therefore accessible in the module subprograms through host association.

### 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.

26 If a procedure declared in the scoping unit of a module has an implicit interface, it shall be given the

- 1 EXTERNAL attribute in that scoping unit; if it is a function, its type and type parameters shall be  
 2 explicitly declared in a type declaration statement in that scoping unit.
- 3 If an intrinsic procedure is declared in the scoping unit of a module, it shall explicitly be given the  
 4 INTRINSIC attribute in that scoping unit or be used as an intrinsic procedure in that scoping unit.

### 5 11.2.1 The USE statement and use association

- 6 The **USE statement** specifies use association. A USE statement is a **module reference** to the module  
 7 it specifies. At the time a USE statement is processed, the public portions of the specified module shall  
 8 be available. A module shall not reference itself, either directly or indirectly. A submodule shall not  
 9 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.

- 10 The **USE statement** provides the means by which a scoping unit accesses named data objects, derived  
 11 types, interface blocks, procedures, abstract interfaces, module procedure interfaces, generic identifiers,  
 12 macros, and namelist groups in a module. The entities in the scoping unit are said to be **use associated**  
 13 with the entities in the module. The accessed entities have the attributes specified in the module, except  
 14 that an entity may have a different accessibility attribute or it may have the ASYNCHRONOUS or  
 15 VOLATILE attribute in the local scoping unit even if the associated module entity does not. The  
 16 entities made accessible are identified by the names or generic identifiers used to identify them in the  
 17 module. By default, they are identified by the same identifiers in the scoping unit containing the USE  
 18 statement, 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).

- |          |                      |           |                                                                                   |
|----------|----------------------|-----------|-----------------------------------------------------------------------------------|
| 19 R1109 | <i>use-stmt</i>      | <b>is</b> | USE [ [ , <i>module-nature</i> ] :: ] <i>module-name</i> [ , <i>rename-list</i> ] |
| 20       |                      | <b>or</b> | USE [ [ , <i>module-nature</i> ] :: ] <i>module-name</i> , ■                      |
| 21       |                      |           | ■ ONLY : [ <i>only-list</i> ]                                                     |
| 22 R1110 | <i>module-nature</i> | <b>is</b> | INTRINSIC                                                                         |
| 23       |                      | <b>or</b> | NON_INTRINSIC                                                                     |
| 24 R1111 | <i>rename</i>        | <b>is</b> | <i>local-name</i> => <i>use-name</i>                                              |
| 25       |                      | <b>or</b> | OPERATOR ( <i>local-defined-operator</i> ) => ■                                   |
| 26       |                      |           | ■ OPERATOR ( <i>use-defined-operator</i> )                                        |
| 27 R1112 | <i>only</i>          | <b>is</b> | <i>generic-spec</i>                                                               |
| 28       |                      | <b>or</b> | <i>only-use-name</i>                                                              |
| 29       |                      | <b>or</b> | <i>rename</i>                                                                     |
| 30 R1113 | <i>only-use-name</i> | <b>is</b> | <i>use-name</i>                                                                   |

- 31 C1108 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.
- 32 C1109 (R1109) If *module-nature* is NON\_INTRINSIC, *module-name* shall be the name of a nonintrinsic  
 33 module.
- 34 C1110 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the

1 same name.

2 C1111 (R1111) OPERATOR(*use-defined-operator*) shall not identify a *generic-binding*.

3 C1112 (R1112) The *generic-spec* shall not identify a *generic-binding*.

#### NOTE 11.9

The above two constraints do not prevent accessing a *generic-spec* that is declared by an interface block, even if a *generic-binding* has the same *generic-spec*.

4 C1113 (R1112) Each *generic-spec* shall be a public entity in the module.

5 C1114 (R1113) Each *use-name* shall be the name of a public entity in the module.

6 R1114 *local-defined-operator*            **is** *defined-unary-op*

7                                            **or** *defined-binary-op*

8 R1115 *use-defined-operator*           **is** *defined-unary-op*

9                                            **or** *defined-binary-op*

10 C1115 (R1115) Each *use-defined-operator* shall be a public entity in the module.

11 A *use-stmt* without a *module-nature* provides access either to an intrinsic or to a nonintrinsic module.  
 12 If the *module-name* is the name of both an intrinsic and a nonintrinsic module, the nonintrinsic module  
 13 is accessed.

14 The USE statement without the ONLY option provides access to all public entities in the specified  
 15 module.

16 A USE statement with the ONLY option provides access only to those entities that appear as *generic-*  
 17 *specs*, *use-names*, or *use-defined-operators* in the *only-list*.

18 More than one USE statement for a given module may appear in a scoping unit. If one of the USE  
 19 statements is without an ONLY option, all public entities in the module are accessible. If all the USE  
 20 statements have ONLY options, only those entities in one or more of the *only-lists* are accessible.

21 An accessible entity in the referenced module has one or more local identifiers. These identifiers are

22        (1) the identifier of the entity in the referenced module if that identifier appears as an *only-use-*  
 23           *name* or as the *defined-operator* of a *generic-spec* in any *only* for that module,

24        (2) each of the *local-names* or *local-defined-operators* that the entity is given in any *rename* for  
 25           that module, and

26        (3) the identifier of the entity in the referenced module if that identifier does not appear as a  
 27           *use-name* or *use-defined-operator* in any *rename* for that module.

28 Two or more accessible entities, other than generic interfaces or defined operators, may have the same  
 29 identifier only if the identifier is not used to refer to an entity in the scoping unit. Generic interfaces  
 30 and defined operators are handled as described in subclause 16.3.4. Except for these cases, the local  
 31 identifier of any entity given accessibility by a USE statement shall differ from the local identifiers of all  
 32 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.

33 The local identifier of an entity made accessible by a USE statement shall not appear in any other

- 1 nonexecutable statement that would cause any attribute (5.3) of the entity to be specified in the scoping  
 2 unit that contains the USE statement, except that it may appear in a PUBLIC or PRIVATE statement  
 3 in the scoping unit of a module and it may be given the ASYNCHRONOUS or VOLATILE attribute.
- 4 The appearance of such a local identifier in a PUBLIC statement in a module causes the entity accessible  
 5 by the USE statement to be a public entity of that module. If the identifier appears in a PRIVATE  
 6 statement in a module, the entity is not a public entity of that module. If the local identifier does not  
 7 appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (5.4.1)  
 8 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.

## 9 11.2.2 Submodules

- 10 A **submodule** is a program unit that extends a module or another submodule. The program unit that  
 11 it extends is its **parent**, and is specified by the *parent-identifier* in the *submodule-stmt*. A submodule  
 12 is a **child** of its parent. An **ancestor** of a submodule is its parent or an ancestor of its parent. A  
 13 **descendant** of a module or submodule is one of its children or a descendant of one of its children. The  
 14 **submodule identifier** is the ordered pair whose first element is the ancestor module name and whose  
 15 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.

1 A submodule accesses the scoping unit of its parent by host association.

2 A submodule may provide implementations for module procedures, each of which is declared by a module  
3 procedure interface body (12.4.2.1) within that submodule or one of its ancestors, and declarations and  
4 definitions of other entities that are accessible by host association in its descendants.

5 R1116 *submodule*                                **is** *submodule-stmt*  
6                                                        [ *specification-part* ]  
7                                                        [ *module-subprogram-part* ]  
8                                                        *end-submodule-stmt*

9 R1117 *submodule-stmt*                           **is** SUBMODULE ( *parent-identifier* ) *submodule-name*

10 R1118 *parent-identifier*                       **is** *ancestor-module-name* [ : *parent-submodule-name* ]

11 R1119 *end-submodule-stmt*                   **is** END [ SUBMODULE [ *submodule-name* ] ]

12 C1116 (R1116) An automatic object shall not appear in the *specification-part* of a submodule.

13 C1117 (R1116) A submodule *specification-part* shall not contain a *format-stmt* or a *stmt-function-stmt*.

14 C1118 (R1116) An object with default initialization that is declared in the *specification-part* of a  
15 submodule shall have the ALLOCATABLE, POINTER, or SAVE attribute.

16 C1119 (R1118) The *ancestor-module-name* shall be the name of a nonintrinsic module; the *parent-*  
17 *submodule-name* shall be the name of a descendant of that module.

18 C1120 (R1116) If a *submodule-name* appears in the *end-submodule-stmt*, it shall be identical to the one  
19 in the *submodule-stmt*.

## 20 **11.3 Block data program units**

21 A **block data program unit** is used to provide initial values for data objects in named common blocks.

22 R1120 *block-data*                                **is** *block-data-stmt*  
23                                                        [ *specification-part* ]  
24                                                        *end-block-data-stmt*

25 R1121 *block-data-stmt*                           **is** BLOCK DATA [ *block-data-name* ]

26 R1122 *end-block-data-stmt*                   **is** END [ BLOCK DATA [ *block-data-name* ] ]

27 C1121 (R1120) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided  
28 in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-*  
29 *data-stmt*.

30 C1122 (R1120) A *block-data specification-part* shall contain only derived-type definitions and ASYN-  
31 CHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRIN-  
32 SIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration  
33 statements.

34 C1123 (R1120) A type declaration statement in a *block-data specification-part* shall not contain AL-



- 1           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.

- 2   If an object in a named common block is initially defined, all storage units in the common block storage  
3   sequence shall be specified even if they are not all initially defined. More than one named common block  
4   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.

- 5   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.

- 6   The same named common block shall not be specified in more than one block data program unit in a  
7   program.  
8   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 2.2.3. 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 (12.5.1). 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 A procedure is classified according to the form of its reference and the way it is defined.

#### 18 12.2.1 Procedure classification by reference

19 The definition of a procedure specifies it to be a function or a subroutine. A reference to a function either  
20 appears explicitly as a primary within an expression, or is implied by a defined operation (7.1.3) within  
21 an expression. A reference to a subroutine is a CALL statement or a defined assignment statement  
22 (7.4.1.4).

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 A procedure is either an intrinsic procedure, an external procedure, a module procedure, an internal  
26 procedure, a dummy procedure (which may be a dummy procedure pointer), a nondummy procedure  
27 pointer, or a statement function.

##### 28 12.2.2.1 Intrinsic procedures

29 A procedure that is provided as an inherent part of the processor is an **intrinsic procedure**.

##### 30 12.2.2.2 External, internal, and module procedures

31 An **external procedure** is a procedure that is defined by an external subprogram or by a means other  
32 than Fortran.

33 An **internal procedure** is a procedure that is defined by an internal subprogram. Internal subprograms  
34 may appear in the main program, in an external subprogram, or in a module subprogram. Internal

1 subprograms shall not appear in other internal subprograms. Internal subprograms are the same as  
2 external subprograms except that the name of the internal procedure is not a global identifier, an  
3 internal subprogram shall not contain an ENTRY statement, the internal procedure name shall not be  
4 argument associated with a dummy procedure (12.5.1.6), and the internal subprogram has access to host  
5 entities by host association.

6 A **module procedure** is a procedure that is defined by a module subprogram.

7 A subprogram defines a procedure for the SUBROUTINE or FUNCTION statement. If the subprogram  
8 has one or more ENTRY statements, it also defines a procedure for each of them.

### 9 12.2.2.3 Dummy procedures

10 A dummy argument that is specified to be a procedure or appears in a procedure reference is a **dummy**  
11 **procedure**. A dummy procedure with the POINTER attribute is a dummy procedure pointer.

### 12 12.2.2.4 Procedure pointers

13 A procedure pointer is a procedure that has the EXTERNAL and POINTER attributes; it may be  
14 pointer associated with an external procedure, a module procedure, an intrinsic procedure, or a dummy  
15 procedure that is not a procedure pointer.

### 16 12.2.2.5 Statement functions

17 A function that is defined by a single statement is a **statement function** (12.6.4).

## 18 12.3 Characteristics of procedures

19 The **characteristics of a procedure** are the classification of the procedure as a function or subroutine,  
20 whether it is pure, whether it is elemental, whether it has the BIND attribute, the characteristics of its  
21 dummy arguments, and the characteristics of its result value if it is a function.

### 22 12.3.1 Characteristics of dummy arguments

23 Each dummy argument has the characteristic that it is a dummy data object, a dummy procedure, a  
24 dummy procedure pointer, or an asterisk (alternate return indicator). A dummy argument other than an asterisk  
25 may be specified to have the OPTIONAL attribute. This attribute means that the dummy argument  
26 need not be associated with an actual argument for any particular reference to the procedure.

#### 27 12.3.1.1 Characteristics of dummy data objects

28 The characteristics of a dummy data object are its type, its type parameters (if any), its shape, its intent  
29 (5.3.9, 5.4.8), whether it is optional (5.3.11, 5.4.9), whether it is allocatable (5.3.7.4), whether it has the  
30 ASYNCHRONOUS (5.3.4), CONTIGUOUS (5.3.6), VALUE (5.3.17), or VOLATILE (5.3.18) attributes,  
31 whether it is polymorphic, and whether it is a pointer (5.3.13, 5.4.11) or a target (5.3.16, 5.4.14). If  
32 a type parameter of an object or a bound of an array is not an initialization expression, the exact  
33 dependence on the entities in the expression is a characteristic. If a shape, size, or type parameter is  
34 assumed or deferred, it is a characteristic.

#### 35 12.3.1.2 Characteristics of dummy procedures and dummy procedure pointers

36 The characteristics of a dummy procedure are the explicitness of its interface (12.4.1), its characteristics  
37 as a procedure if the interface is explicit, whether it is a pointer, and whether it is optional (5.3.11, 5.4.9).

### 1 12.3.1.3 Characteristics of asterisk dummy arguments

2 An asterisk as a dummy argument has no characteristics.

### 3 12.3.2 Characteristics of function results

4 The characteristics of a function result are its type, type parameters (if any), rank, whether it is poly-  
 5 morphic, whether it is allocatable, whether it is a pointer, whether it has the CONTIGUOUS attribute,  
 6 and whether it is a procedure pointer. If a function result is an array that is not allocatable or a pointer,  
 7 its shape is a characteristic. If a type parameter of a function result or a bound of a function result  
 8 array is not an initialization expression, the exact dependence on the entities in the expression is a  
 9 characteristic. If type parameters of a function result are deferred, which parameters are deferred is a  
 10 characteristic. If the length of a character function result is assumed, this is a characteristic.

## 11 12.4 Procedure interface

12 The **interface** of a procedure determines the forms of reference through which it may be invoked.  
 13 The procedure's interface consists of its abstract interface, its name, its binding label if any, and the  
 14 procedure's generic identifiers, if any. The characteristics of a procedure are fixed, but the remainder  
 15 of the interface may differ in different scoping units, except that for a separate module procedure body  
 16 (12.6.2.4), the dummy argument names, binding label, and whether it is recursive shall be the same as  
 17 in its corresponding module procedure interface body (12.4.2.1).

18 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.

### 19 12.4.1 Implicit and explicit interfaces

20 If a procedure is accessible in a scoping unit, its interface is either **explicit** or **implicit** in that scoping  
 21 unit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit  
 22 in such a scoping unit. The interface of a subroutine or a function with a separate result name is explicit  
 23 within the subprogram that defines it. The interface of a statement function is always implicit. The interface of  
 24 an external procedure or dummy procedure is explicit in a scoping unit other than its own if an interface  
 25 body (12.4.2.1) 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.

#### 26 12.4.1.1 Explicit interface

27 A procedure other than a statement function shall have an explicit interface if it is referenced and

- 28 (1) a reference to the procedure appears
  - 29 (a) with an argument keyword (12.5.1),
  - 30 (b) with an argument that is a co-indexed object (2.4.6),

**J3 internal note****Unresolved Technical Issue 49**

This makes most old (F77-style) procedures useless for operating on co-indexed objects. I'm not convinced it is necessary.

I was told: "Surely it makes sense to prevent programming errors when possible."

However, this achieves it not.

Forcing the user to lie about his INTENT, use error-suppressing compiler switches, or forego the use of a third-party library which has unspecified intent arguments (in other words, the vast majority of software) does not prevent programming errors, it encourages them.

- 1 (c) as a reference by its generic name (12.4.2.1),
- 2 (d) as a defined assignment (subroutines only),
- 3 (e) in an expression as a defined operator (functions only), or
- 4 (f) in a context that requires it to be pure,
- 5 (2) the procedure has a dummy argument that
  - 6 (a) has the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, TARGET,
  - 7 VALUE, or VOLATILE attribute,
  - 8 (b) is an assumed-shape array,
  - 9 (c) is a co-array,
  - 10 (d) is of a parameterized derived type, or
  - 11 (e) is polymorphic,
- 12 (3) the procedure has a result that
  - 13 (a) is an array,
  - 14 (b) is a pointer or is allocatable, or
  - 15 (c) has a nonassumed type parameter value that is not an initialization expression,
- 16 (4) the procedure is elemental, or
- 17 (5) the procedure has the BIND attribute.

**12.4.2 Specification of the procedure interface**

19 The interface for an internal, external, module, or dummy procedure is specified by a FUNCTION,  
 20 SUBROUTINE, or ENTRY statement and by specification statements for the dummy arguments and  
 21 the result of a function. These statements may appear in the procedure definition, in an interface body,  
 22 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 (12.4.2.1).

**12.4.2.1 Interface block**

24	R1201	<i>interface-block</i>	<b>is</b>	<i>interface-stmt</i>
25				[ <i>interface-specification</i> ] ...
26				<i>end-interface-stmt</i>
27	R1202	<i>interface-specification</i>	<b>is</b>	<i>interface-body</i>
28			<b>or</b>	<i>procedure-stmt</i>
29	R1203	<i>interface-stmt</i>	<b>is</b>	INTERFACE [ <i>generic-spec</i> ]
30			<b>or</b>	ABSTRACT INTERFACE

- 1 R1204 *end-interface-stmt* is END INTERFACE [ *generic-spec* ]  
2 R1205 *interface-body* is *function-stmt*  
3 [ *specification-part* ]  
4 *end-function-stmt*  
5 or *subroutine-stmt*  
6 [ *specification-part* ]  
7 *end-subroutine-stmt*  
8 R1206 *procedure-stmt* is [ MODULE ] PROCEDURE *procedure-name-list*  
9 R1207 *generic-spec* is *generic-name*  
10 or OPERATOR ( *defined-operator* )  
11 or ASSIGNMENT ( = )  
12 or *dtio-generic-spec*  
13 R1208 *dtio-generic-spec* is READ (FORMATTED)  
14 or READ (UNFORMATTED)  
15 or WRITE (FORMATTED)  
16 or WRITE (UNFORMATTED)  
17 R1209 *import-stmt* is IMPORT [ [ :: ] *import-name-list* ]
- 18 C1201 (R1201) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure  
19 defined by that subprogram.
- 20 C1202 (R1201) The *generic-spec* shall be included in the *end-interface-stmt* only if it is provided in the  
21 *interface-stmt*. If the *end-interface-stmt* includes *generic-name*, the *interface-stmt* shall specify  
22 the same *generic-name*. If the *end-interface-stmt* includes ASSIGNMENT(=), the *interface-*  
23 *stmt* shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *dtio-generic-spec*,  
24 the *interface-stmt* shall specify the same *dtio-generic-spec*. If the *end-interface-stmt* includes  
25 OPERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If  
26 one *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the  
27 corresponding operator <, <=, >, >=, ==, or /=.
- 28 C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the  
29 *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword  
30 that specifies an intrinsic type.
- 31 C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.
- 32 C1205 (R1205) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments  
33 except pointer, alternate return, and procedure arguments.
- 34 C1206 (R1205) An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-*  
35 *function-stmt*.
- 36 C1207 (R1206) A *procedure-name* shall have an explicit interface and shall refer to an accessible pro-  
37 cedure pointer, external procedure, dummy procedure, or module procedure.
- 38 C1208 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall  
39 be accessible in the current scope as a module procedure.
- 40 C1209 (R1206) A *procedure-name* shall not specify a procedure that is specified previously in any  
41 *procedure-stmt* in any accessible interface with the same generic identifier.
- 42 C1210 (R1209) The IMPORT statement is allowed only in an *interface-body* that is not a module  
43 procedure interface body.
- 44 C1211 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.
- 45 An external or module subprogram specifies a **specific interface** for the procedures defined in that

1 subprogram. Such a specific interface is explicit for module procedures and implicit for external proce-  
 2 dures.

3 An interface block introduced by ABSTRACT INTERFACE is an **abstract interface block**. An  
 4 interface body in an abstract interface block specifies an abstract interface. An interface block with a  
 5 generic specification is a **generic interface block**. An interface block with neither ABSTRACT nor a  
 6 generic specification is a **specific interface block**.

7 The name of the entity declared by an interface body is the *function-name* in the *function-stmt* or the  
 8 *subroutine-name* in the *subroutine-stmt* that begins the interface body.

9 A **module procedure interface body** is an interface body whose initial statement contains the  
 10 keyword MODULE. It defines the **module procedure interface** for a separate module procedure  
 11 (12.6.2.4). A separate module procedure is accessible by use association if and only if its interface body  
 12 is declared in the specification part of a module and is public. If a corresponding (12.6.2.4) separate  
 13 module procedure is not defined, the interface may be used to specify an explicit specific interface but  
 14 the procedure shall not be used in any other way.

15 C1212 (R1205) A module procedure interface body shall not appear in an abstract interface block.

16 An interface body in a generic or specific interface block specifies the EXTERNAL attribute and an  
 17 explicit specific interface for an external procedure or a dummy procedure. If the name of the declared  
 18 procedure is that of a dummy argument in the subprogram containing the interface body, the procedure  
 19 is a dummy procedure; otherwise, it is an external procedure.

20 An interface body specifies all of the characteristics of the explicit specific interface or abstract interface.  
 21 The specification part of an interface body may specify attributes or define values for data entities that  
 22 do not determine characteristics of the procedure. Such specifications have no effect.

23 If an explicit specific interface is specified by an interface body or a procedure declaration statement  
 24 (12.4.2.3) for an external procedure, the characteristics shall be consistent with those specified in the  
 25 procedure definition, except that the interface may specify a procedure that is not pure if the procedure  
 26 is defined to be pure. An interface for a procedure named by an ENTRY statement may be specified by  
 27 using the entry name as the procedure name in the interface body. An explicit specific interface may  
 28 be specified by an interface body for an external procedure that does not exist in the program if the  
 29 procedure is never used in any way. A procedure shall not have more than one explicit specific interface  
 30 in a given scoping unit.

#### NOTE 12.4

The dummy argument names may be different because the name of a dummy argument is not a characteristic.

31 The IMPORT statement specifies that the named entities from the host scoping unit are accessible in  
 32 the interface body by host association. An entity that is imported in this manner and is defined in the  
 33 host scoping unit shall be explicitly declared prior to the interface body. The name of an entity made  
 34 accessible by an IMPORT statement shall not appear in any of the contexts described in 16.5.1.4 that  
 35 cause the host entity of that name to be inaccessible.

36 Within an interface body, if an IMPORT statement with no *import-name-list* appears, each host entity  
 37 not named in an IMPORT statement also is made accessible by host association if its name does not  
 38 appear in any of the contexts described in 16.5.1.4 that cause the host entity of that name to be  
 39 inaccessible. If an entity that is made accessible by this means is accessed by host association and is  
 40 defined in the host scoping unit, it shall be explicitly declared prior to the interface body.



**NOTE 12.5**

An example of an interface block without a generic specification 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 (12.5.1); for example:

```
EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

**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.

- 1 A generic interface block specifies a **generic interface** for each of the procedures in the interface

- 1 block. The PROCEDURE statement lists procedure pointers, external procedures, dummy procedures,  
 2 or module procedures that have this generic interface. A generic interface is always explicit.
- 3 Any procedure may be referenced via its specific interface if the specific interface is accessible. It also  
 4 may be referenced via a generic interface. The *generic-spec* in an *interface-stmt* is a **generic identifier**  
 5 for all the procedures in the interface block. The rules specifying how any two procedures with the same  
 6 generic identifier shall differ are given in 16.3.4. They ensure that any generic invocation applies to at  
 7 most one specific procedure.
- 8 A **generic name** specifies a single name to reference all of the procedure names in the interface block.  
 9 A generic name may be the same as any one of the procedure names in the interface block, or the same  
 10 as any accessible generic name.
- 11 A generic name may be the same as a derived-type name, in which case all of the procedures in the  
 12 interface block shall be functions.

#### 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
```

- 13 An *interface-stmt* having a *dtio-generic-spec* is an interface for a user-defined derived-type input/output  
 14 procedure (9.5.3.7)

#### 15 12.4.2.1.1 Defined operations

- 16 If OPERATOR is specified in a generic specification, all of the procedures specified in the generic interface  
 17 shall be functions that may be referenced as defined operations (7.1.3, 7.1.8.8, 7.2, 12.5). In the case of  
 18 functions of two arguments, infix binary operator notation is implied. In the case of functions of one  
 19 argument, prefix operator notation is implied. OPERATOR shall not be specified for functions with no  
 20 arguments or for functions with more than two arguments. The dummy arguments shall be nonoptional  
 21 dummy data objects and shall be specified with INTENT (IN). The function result shall not have assumed  
 22 character length. If the operator is an *intrinsic-operator* (R310), the number of function arguments shall  
 23 be consistent with the intrinsic uses of that operator, and the types, kind type parameters, or ranks of  
 24 the dummy arguments shall differ from those required for the intrinsic operation (7.1.2).

- 25 A defined operation is treated as a reference to the function. For a unary defined operation, the operand  
 26 corresponds to the function's dummy argument; for a binary operation, the left-hand operand corre-

- 1 sponds to the first dummy argument of the function and the right-hand operand corresponds to the  
 2 second dummy argument.

**NOTE 12.8**

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR ( * )
  FUNCTION BOOLEAN_AND (B1, B2)
    LOGICAL, INTENT (IN) :: B1 (:), B2 (SIZE (B1))
    LOGICAL :: BOOLEAN_AND (SIZE (B1))
  END FUNCTION BOOLEAN_AND
END INTERFACE OPERATOR ( * )
```

This allows, for example

```
SENSOR (1:N) * ACTION (1:N)
```

as an alternative to the function call

```
BOOLEAN_AND (SENSOR (1:N), ACTION (1:N))    ! SENSOR and ACTION are
                                              ! of type LOGICAL
```

- 3 A given defined operator may, as with generic names, apply to more than one function, in which case  
 4 it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic  
 5 properties include the intrinsic operations they represent. Because both forms of each relational operator  
 6 have the same interpretation (7.2), extending one form (such as <=) has the effect of defining both forms  
 7 (<= 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 (16.3.4), so it is not possible to define a generic operator that has one procedure for pointers and another procedure for nonpointers.

#### 8 12.4.2.1.2 Defined assignments

- 9 If ASSIGNMENT ( = ) is specified in a generic specification, all the procedures in the generic interface  
 10 shall be subroutines that may be referenced as defined assignments (7.4.1.4). Defined assignment may,  
 11 as with generic names, apply to more than one subroutine, in which case it is generic in exact analogy  
 12 to generic procedure names.
- 13 Each of these subroutines shall have exactly two dummy arguments. The dummy arguments shall be  
 14 nonoptional dummy data objects. The first argument shall have INTENT (OUT) or INTENT (INOUT)  
 15 and the second argument shall have INTENT (IN). Either the second argument shall be an array whose  
 16 rank differs from that of the first argument, the declared types and kind type parameters of the arguments  
 17 shall not conform as specified in Table 7.10, or the first argument shall be of derived type. A defined  
 18 assignment is treated as a reference to the subroutine, with the left-hand side as the first argument  
 19 and the right-hand side enclosed in parentheses as the second argument. The ASSIGNMENT generic

- 1 specification specifies that assignment is extended or redefined.

#### NOTE 12.10

An example of the use of the ASSIGNMENT generic specification is:

```
INTERFACE ASSIGNMENT ( = )

  SUBROUTINE LOGICAL_TO_NUMERIC (N, B)
    INTEGER, INTENT (OUT) :: N
    LOGICAL, INTENT (IN)  :: B
  END SUBROUTINE LOGICAL_TO_NUMERIC
  SUBROUTINE CHAR_TO_STRING (S, C)
    USE STRING_MODULE      ! Contains definition of type STRING
    TYPE (STRING), INTENT (OUT) :: S ! A variable-length string
    CHARACTER (*), INTENT (IN)  :: C
  END SUBROUTINE CHAR_TO_STRING
END INTERFACE ASSIGNMENT ( = )
```

Example assignments are:

```
KOUNT = SENSOR (J) ! CALL LOGICAL_TO_NUMERIC (KOUNT, (SENSOR (J)))
NOTE  = '89AB'     ! CALL CHAR_TO_STRING (NOTE, ('89AB'))
```

#### 2 12.4.2.1.3 User-defined derived-type input/output procedure interfaces

- 3 All of the procedures specified in an interface block for a user-defined derived-type input/output proce-  
4 dure shall be subroutines that have interfaces as described in 9.5.3.7.2.

#### 5 12.4.2.2 EXTERNAL statement

- 6 An **EXTERNAL statement** specifies the EXTERNAL attribute (5.3.8) for a list of names.

7 R1210 *external-stmt*                      **is**    EXTERNAL [ :: ] *external-name-list*

- 8 The appearance of the name of a block data program unit in an EXTERNAL statement confirms that  
9 the block data program unit is a part of the program.

#### NOTE 12.11

For explanatory information on potential portability problems with external procedures, see sub-  
clause C.9.1.

#### NOTE 12.12

An example of an EXTERNAL statement is:

```
EXTERNAL FOCUS
```

#### 10 12.4.2.3 Procedure declaration statement

- 11 A procedure declaration statement declares procedure pointers, dummy procedures, and external pro-  
12 cedures. It specifies the EXTERNAL attribute (5.3.8) for all procedure entities in the *proc-decl-list*.

13 R1211 *procedure-declaration-stmt*    **is**    PROCEDURE ( [ *proc-interface* ] ) ■  
14                                              ■ [ [ , *proc-attr-spec* ] ... :: ] *proc-decl-list*

1 R1212 *proc-interface* is *interface-name*  
 2 or *declaration-type-spec*  
 3 R1213 *proc-attr-spec* is *access-spec*  
 4 or *proc-language-binding-spec*  
 5 or INTENT ( *intent-spec* )  
 6 or OPTIONAL  
 7 or POINTER  
 8 or SAVE  
 9 R1214 *proc-decl* is *procedure-entity-name*[ => *proc-pointer-init* ]  
 10 R1215 *interface-name* is *name* R1216 *proc-pointer-init* is *null-init*  
 11 or *initial-proc-target*  
 12 R1217 *initial-proc-target* is *procedure-name*  
 13

14 C1213 (R1215) The *name* shall be the name of an abstract interface or of a procedure that has an  
 15 explicit interface. If *name* is declared by a *procedure-declaration-stmt* it shall be previously  
 16 declared. If *name* denotes an intrinsic procedure it shall be one that is listed in 13.6 and not  
 17 marked with a bullet (●).

18 C1214 (R1215) The *name* shall not be the same as a keyword that specifies an intrinsic type.

19 C1215 If a procedure entity has the INTENT attribute or SAVE attribute, it shall also have the  
 20 POINTER attribute.

21 C1216 (R1211) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall  
 22 specify an external procedure.

23 C1217 (R1214) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.

24 C1218 (R1217) The *procedure-name* shall be the name of an initialization target.

25 C1219 (R1211) If *proc-language-binding-spec* with a NAME= is specified, then *proc-decl-list* shall con-  
 26 tain exactly one *proc-decl*, which shall neither have the POINTER attribute nor be a dummy  
 27 procedure.

28 C1220 (R1211) If *proc-language-binding-spec* is specified, the *proc-interface* shall appear, it shall be an  
 29 *interface-name*, and *interface-name* shall be declared with a *proc-language-binding-spec*.

30 If *proc-interface* appears and consists of *interface-name*, it specifies an explicit specific interface (12.4.2.1)  
 31 for the declared procedures or procedure pointers. The abstract interface (12.4) is that specified by the  
 32 interface named by *interface-name*.

33 If *proc-interface* appears and consists of *declaration-type-spec*, it specifies that the declared procedures  
 34 or procedure pointers are functions having implicit interfaces and the specified result type. If a type is  
 35 specified for an external function, its function definition (12.6.2.1) shall specify the same result type and  
 36 type parameters.

37 If *proc-interface* does not appear, the procedure declaration statement does not specify whether the  
 38 declared procedures or procedure pointers are subroutines or functions.

39 If a *proc-attr-spec* other than a *proc-language-binding-spec* appears, it specifies that the declared proce-  
 40 dures or procedure pointers have that attribute. These attributes are described in 5.3. If a *proc-language-*  
 41 *binding-spec* with NAME= appears, it specifies a binding label or its absence, as described in 15.5.2.  
 42 A *proc-language-binding-spec* without a NAME= is allowed, but is redundant with the *proc-interface*  
 43 required by C1220.

44 A procedure is an initialization target if it is a nonelemental external or module procedure, or a specific

- 1 intrinsic function listed in 13.6 and not marked with a bullet (●).
- 2 If `=>` appears in a *proc-decl* in a *procedure-declaration-stmt* it specifies the initial association status
- 3 of the corresponding procedure entity, and implies the SAVE attribute, which may be confirmed by
- 4 explicit specification. If `=> null-init` appears, the procedure entity is initially disassociated. If `=>`
- 5 *initial-proc-target* appears, the procedure entity is initially associated with the target.
- 6 If *proc-entity-name* has an explicit interface, its characteristics shall be the same as *initial-proc-target*
- 7 except that *initial-proc-target* may be pure even if *proc-entity-name* is not pure and *initial-proc-target*
- 8 may be an elemental intrinsic procedure.
- 9 If the characteristics of *proc-entity-name* or *initial-proc-target* are such that an explicit interface is
- 10 required, both *proc-entity-name* and *initial-proc-target* shall have an explicit interface.
- 11 If *proc-entity-name* has an implicit interface and is explicitly typed or referenced as a function, *initial-*
- 12 *proc-target* shall be a function. If *proc-entity-name* has an implicit interface and is referenced as a
- 13 subroutine, *initial-proc-target* shall be a subroutine.
- 14 If *initial-proc-target* and *proc-entity-name* are functions, they shall have the same type; corresponding
- 15 type parameters shall either both be deferred or both have the same value.

**NOTE 12.13**

In contrast to the EXTERNAL statement, it is not possible to use the PROCEDURE statement to identify a BLOCK DATA subprogram.

**NOTE 12.14**

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)
    REAL, INTENT (IN) :: X
  END SUBROUTINE SUB
END INTERFACE

!-- Some external or dummy procedures with explicit interface.
PROCEDURE (REAL_FUNC) :: BESSEL, GAMMA
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_GAMMA
!-- 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.
```

**NOTE 12.14 (cont.)**

```

TYPE(STRUCT_TYPE) :: STRUCT
!-- An external or dummy function with implicit interface
PROCEDURE (REAL) :: PSI

```

**1 12.4.2.4 INTRINSIC statement**

2 An **INTRINSIC statement** specifies a list of names that have the INTRINSIC attribute (5.3.10).

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 12.15**

A name shall not be explicitly specified to have both the EXTERNAL and INTRINSIC attributes in the same scoping unit.

**5 12.4.2.5 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 The form of a procedure reference is dependent on the interface of the procedure or procedure pointer,  
13 but is independent of the means by which the procedure is defined. The forms of procedure references  
14 are:

15 R1219 *function-reference*                      **is** *procedure-designator* ( [ *actual-arg-spec-list* ] )

16 C1222 (R1219) The *procedure-designator* shall designate a function.

17 C1223 (R1219) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.

18 R1220 *call-stmt*                                      **is** CALL *procedure-designator* [ ( [ *actual-arg-spec-list* ] ) ]

19 C1224 (R1220) The *procedure-designator* shall designate a subroutine.

20 R1221 *procedure-designator*                      **is** *procedure-name*  
21                                                              **or** *proc-component-ref*  
22                                                              **or** *data-ref* % *binding-name*

23 C1225 (R1221) A *procedure-name* shall be the name of a procedure or procedure pointer.

24 C1226 (R1221) A *binding-name* shall be a binding name (4.5.5) of the declared type of *data-ref*.

25 C1227 (R1221) If *data-ref* is an array, the referenced type-bound procedure shall have the PASS at-  
26 tribute.

27 Resolving references to type-bound procedures is described in 12.5.5.

28 A function may also be referenced as a defined operation (12.4.2.1.1). A subroutine may also be referenced

- 1 as a defined assignment (12.4.2.1.2).

#### NOTE 12.16

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*.

2 R1222 *actual-arg-spec*                    **is** [ *keyword* = ] *actual-arg*  
 3 R1223 *actual-arg*                        **is** *expr*  
 4                                            **or** *variable*  
 5                                            **or** *procedure-name*  
 6                                            **or** *proc-component-ref*  
 7                                            **or** *alt-return-spec*  
 8 R1224 *alt-return-spec*                   **is** \* *label*

9 C1228 (R1222) The *keyword* = shall not appear if the interface of the procedure is implicit in the  
 10 scoping unit.

11 C1229 (R1222) The *keyword* = shall not be omitted from an *actual-arg-spec* unless it has been omitted  
 12 from each preceding *actual-arg-spec* in the argument list.

13 C1230 (R1222) Each *keyword* shall be the name of a dummy argument in the explicit interface of the  
 14 procedure.

15 C1231 (R1223) A nonintrinsic elemental procedure shall not be used as an actual argument.

16 C1232 (R1223) A *procedure-name* shall be the name of an external, internal, module, or dummy proce-  
 17 dure, a specific intrinsic function listed in 13.6 and not marked with a bullet (•), or a procedure  
 18 pointer.

19 C1233 (R1223) In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a  
 20 pure procedure (12.7).

#### NOTE 12.17

This constraint ensures that the purity of a procedure cannot be undermined by allowing it to call a nonpure procedure.

21 C1234 (R1224) The *label* shall be the statement label of a branch target statement that appears in the same scoping  
 22 unit as the *call-stmt*.

#### NOTE 12.18

Successive commas shall not be used to omit optional arguments.

#### NOTE 12.19

Examples of procedure reference using procedure pointers:

```
P => BESSEL
WRITE (*, *) P(2.5)      !-- BESSEL(2.5)

S => PRINT_REAL
CALL S(3.14)
```



## 12.5.1 Actual arguments, dummy arguments, and argument association

### 12.5.1.1 Argument correspondence

In either a subroutine reference or a function reference, the actual argument list identifies the correspondence between the actual arguments supplied and the dummy arguments of the procedure. This correspondence may be established either by keyword or by position. If an argument keyword appears, the actual argument corresponds to the dummy argument whose name is the same as the argument keyword (using the dummy argument names from the interface accessible in the scoping unit containing the procedure reference). In the absence of an argument keyword, an actual argument corresponds to the dummy argument occupying the corresponding position in the reduced dummy argument list; that is, the first actual argument corresponds to the first dummy argument in the reduced list, the second actual argument corresponds to the second dummy argument in the reduced list, etc. The reduced dummy argument list is either the full dummy argument list or, if there is a passed-object dummy argument (4.5.4.4), the dummy argument list with the passed object dummy argument omitted. Exactly one actual argument shall correspond to each nonoptional dummy argument. At most one actual argument shall correspond to each optional dummy argument. Each actual argument shall correspond to a dummy argument.

#### NOTE 12.20

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.

### 12.5.1.2 The passed-object dummy argument and argument correspondence

In a reference to a type-bound procedure, or a procedure pointer component, that has a passed-object dummy argument (4.5.4.4), the *data-ref* of the *function-reference* or *call-stmt* corresponds, as an actual argument, with the passed-object dummy argument.

### 12.5.1.3 Argument association

Except in references to intrinsic inquiry functions, if a nonoptional nonpointer dummy argument corresponds to a pointer actual argument, the actual argument shall be pointer associated with a target and the dummy argument becomes argument associated with that target. If an optional nonpointer dummy argument corresponds to a pointer actual argument that is pointer associated with a target the dummy argument becomes argument associated with that target. A present nonpointer dummy argument that corresponds to a nonpointer actual argument becomes argument associated with that actual argument.

- 1 A present pointer dummy argument that corresponds to a pointer actual argument becomes argument  
2 associated with that actual argument. A present pointer dummy argument that does not correspond to  
3 a pointer actual argument is not argument associated.

#### 4 12.5.1.4 Actual arguments associated with dummy data objects

- 5 If a dummy argument is neither allocatable nor a pointer, it shall be type compatible or bits compatible  
6 (4.3.1.3) with the associated actual argument. If a dummy argument is allocatable or a pointer, the  
7 associated actual argument shall be polymorphic if and only if the 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 12.21

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.

- 10 Unless the actual argument and the corresponding dummy argument are bits compatible, the type  
11 parameter values of the actual argument shall agree with the corresponding ones of the dummy argument  
12 that are not assumed or deferred, except for the case of the character length parameter of an actual  
13 argument of type default character associated with a dummy argument that is not assumed shape.
- 14 If a scalar dummy argument is of type default character, the length *len* of the dummy argument shall  
15 be less than or equal to the length of the actual argument. The dummy argument becomes associated  
16 with the leftmost *len* characters of the actual argument. If an array dummy argument is of type default  
17 character and is not assumed shape, it becomes associated with the leftmost characters of the actual  
18 argument element sequence (12.5.1.8) and it shall not extend beyond the end of that sequence.
- 19 The values of assumed type parameters of a dummy argument are assumed from the corresponding type  
20 parameters of the associated actual argument.
- 21 An actual argument associated with a dummy argument that is allocatable or a pointer shall have  
22 deferred the same type parameters as the dummy argument.
- 23 If the actual argument is a co-indexed object, the dummy argument shall not be a co-array and shall  
24 have the INTENT(IN) or the VALUE attribute.

**J3 internal note**

Unresolved Technical Issue 50

It doesn't seem like forcing the user to write

```
localvar = x[i]
CALL sub(localvar)
x[i] = localvar
```

is a substantial improvement in, well, anything. Not to mention that the user cannot do that if `x[i]` is undefined or only partially defined before the call to `sub`. It doesn't seem to be any more work for the processor than what it already does for

```
CALL sub(a(1:n:2))
```

for explicit-shape dummies. And, *MUCH WORSE*, this makes defined assignment unusable with co-arrays. Can we say "not even integrated with Fortran 90"?

Subgroup said: "this requires more complications to the memory consistency rules since the copy out may overwrite other changes to the variable."

The editor says: That is just so untrue it is not funny.

Is everyone *REALLY* that ignorant of our dummy argument aliasing rules?

We've not repealed them, and there are no edits to subvert them for co-arrays. If they need to be subverted (something I doubt, but maybe the go-faster department want to go slower), something needs to be done.

Either way this is a serious technical flaw.

**NOTE 12.22**

If the actual argument is a co-indexed object and the corresponding dummy argument is not a co-array, it is likely that a processor will make a copy on the executing image of the actual argument, including copies of any allocated allocatable subcomponents, before argument association occurs.

**J3 internal note**

Unresolved Technical Issue 51

That doesn't seem likely for the vast majority of Fortran processing systems, which are going to be shared-memory small-cpu-count. They are, presumably, going to do the "right thing" here.

That does not involve making any copies.

Anyway, what's this "and the ... dummy is not a co-array." guff. Last time I looked, a co-indexed object was not a valid actual argument to a co-array dummy.

- 1 If the dummy argument is a pointer that does not have the `INTENT(IN)` attribute, the actual argument
- 2 shall be a pointer. Otherwise, the actual argument shall be a pointer or a valid target for the dummy
- 3 pointer in an assignment statement. If the actual argument is not a pointer, the dummy pointer becomes
- 4 pointer-associated with the actual argument.
- 5 If the dummy argument and actual argument are pointers, the ranks shall agree and either the non-
- 6 deferred type parameters shall agree or the actual argument and the dummy argument shall be bits
- 7 compatible. If a dummy pointer has the `CONTIGUOUS` attribute, the actual argument shall have the
- 8 `CONTIGUOUS` attribute.

**J3 internal note****Unresolved Technical Issue 52**

Argument-associating pointers of different types does not seem to be in accordance with the spirit of our current rules. It requires a processor to use the same representation for pointers of different kinds or to do copyin/copyout conversions. Why are we doing this?

Furthermore, this “pseudo-polymorphism” for bits leads directly to INTEGER pointers being associated with REAL targets. This will be a serious inhibition on optimisation. It is an open invitation to programming errors and nonportable hacks.

NOTE: A similar comment would apply to pointer assignment if one is allowed to turn a bits pointer into another pointer (the term “bits compatible” not being well-defined as yet, I don’t know whether this is the case). Fiddling with the bits is one thing, but removing the type restrictions on our pointers is simply unacceptable both from a software engineering point of view and from a performance point of view.

- 1 If a dummy argument is allocatable, the actual argument shall be allocatable, the ranks shall agree, and
- 2 either the nondeferred type parameters shall agree or the actual argument and the dummy argument
- 3 shall be bits compatible. It is permissible for the actual argument to have an allocation status of
- 4 unallocated.

**J3 internal note****Unresolved Technical Issue 53**

Argument-associating allocatables of different types does not seem to be in accordance with the spirit of our current rules. It requires a processor to use the same representation for allocatables of different kinds or to do copyin/copyout conversions. Why are we doing this?

Furthermore, the very concept of having allocatable entities of REAL and INTEGER types being associated with one another is blood-curdling. Just how far the implications of this would go I am not sure; certainly they are not good for reliability and portability, whether performance suffers as well I cannot say without spending considerable effort.

This butchery of our association rules does not seem at all necessary for BITS to be useful.

- 5 At the invocation of the procedure, the pointer association status of an actual argument associated with
- 6 a pointer dummy argument becomes undefined if the dummy argument has INTENT(OUT).
- 7 Except in references to intrinsic inquiry functions, if a nonoptional dummy argument is not allocatable
- 8 and the actual argument is allocatable, the corresponding actual argument shall be allocated.
- 9 If the dummy argument has the VALUE attribute it becomes associated with a definable anonymous
- 10 data object whose initial value is that of the actual argument. Subsequent changes to the value or
- 11 definition status of the dummy argument do not affect the actual argument.

**NOTE 12.23**

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. The actual mechanism by which this happens is determined by the processor.

- 12 If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with
- 13 the actual argument do not become associated with the corresponding dummy argument on invocation
- 14 of the procedure. If such a dummy argument is used as an actual argument that is associated with
- 15 a dummy argument with the TARGET attribute, whether any pointers associated with the original
- 16 actual argument become associated with the dummy argument with the TARGET attribute is processor
- 17 dependent.
- 18 If the dummy argument has the TARGET attribute, does not have the VALUE attribute, and is either a

1 scalar or an assumed-shape array that does not have the CONTIGUOUS attribute, and the corresponding  
 2 actual argument has the TARGET attribute but is not a co-indexed object or an array section with a  
 3 vector subscript then

- 4 (1) any pointers associated with the actual argument become associated with the corresponding  
 5 dummy argument on invocation of the procedure, and
- 6 (2) when execution of the procedure completes, any pointers that do not become undefined  
 7 (16.5.2.2.3) and are associated with the dummy argument remain associated with the actual  
 8 argument.

9 If the dummy argument has the TARGET attribute and is an explicit-shape array, an assumed-shape ar-  
 10 ray with the CONTIGUOUS attribute, or an assumed-size array, and the corresponding actual argument  
 11 has the TARGET attribute but is not an array section with a vector subscript then

- 12 (1) on invocation of the procedure, whether any pointers associated with the actual argument  
 13 become associated with the corresponding dummy argument is processor dependent, and
- 14 (2) when execution of the procedure completes, the pointer association status of any pointer  
 15 that is pointer associated with the dummy argument is processor dependent.

16 If the dummy argument has the TARGET attribute and the corresponding actual argument does not  
 17 have the TARGET attribute or is an array section with a vector subscript, any pointers associated with  
 18 the dummy argument become undefined when execution of the procedure completes.

19 If the dummy argument has the TARGET attribute and the VALUE attribute, any pointers associated  
 20 with the dummy argument become undefined when execution of the procedure completes.

21 If the actual argument is scalar, the corresponding dummy argument shall be scalar unless the actual  
 22 argument is of type default character, of type character with the C character kind (15.2), or is an element  
 23 or substring of an element of an array that is not an assumed-shape, pointer, or polymorphic array. If  
 24 the procedure is nonelemental and is referenced by a generic name or as a defined operator or defined  
 25 assignment, the ranks of the actual arguments and corresponding dummy arguments shall agree.

26 If a dummy argument is an assumed-shape array, the rank of the actual argument shall be the same as  
 27 the rank of the dummy argument; the actual argument shall not be an assumed-size array (including an  
 28 array element designator or an array element substring designator).

29 Except when a procedure reference is elemental (12.8), each element of an array actual argument or of  
 30 a sequence in a sequence association (12.5.1.8) is associated with the element of the dummy array that  
 31 has the same position in array element order (6.2.2.2).

#### NOTE 12.24

For type default character sequence associations, the interpretation of element is provided in 12.5.1.8.

32 A scalar dummy argument of a nonelemental procedure may be associated only with a scalar actual  
 33 argument.

34 If a nonpointer dummy argument has INTENT (OUT) or INTENT (INOUT), the argument associated  
 35 entity shall be definable. If a dummy argument has INTENT (OUT), the argument associated entity  
 36 becomes undefined at the time the association is established, except for components of an object of de-  
 37 rived type for which default initialization has been specified. If the dummy argument is not polymorphic  
 38 and the type of the actual argument is an extension type of the type of the dummy argument, only the  
 39 part of the actual argument that is of the same type as the dummy argument becomes undefined.

40 If the actual argument is an array section having a vector subscript, the dummy argument is not defin-  
 41 able and shall not have the INTENT (OUT), INTENT (INOUT), VOLATILE, or ASYNCHRONOUS

1 attributes.

#### NOTE 12.25

Argument intent specifications serve several purposes. See Note 5.16.

#### NOTE 12.26

For more explanatory information on argument association and evaluation, see subclause C.9.5. For more explanatory information on pointers and targets as dummy arguments, see subclause C.9.6.

#### J3 internal note

Unresolved Technical Issue 55

Probably need this constraint:

C1235 An actual argument that is a co-indexed object shall not be associated with a dummy argument that has the ASYNCHRONOUS attribute.

It is true that allowing VALUE+ASYNCHRONOUS makes a mockery of the restrictions on ASYNCHRONOUS dummies and note (04-007) 12.26 about their purpose. But should co-arrays try to be consistent or should they copy this inconsistency?

2 C1236 (R1223) If an actual argument is an array section or an assumed-shape array, and the corre-  
3 sponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that  
4 dummy argument shall be an assumed-shape array.

5 C1237 (R1223) If an actual argument is a pointer array, and the corresponding dummy argument  
6 has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an  
7 assumed-shape array that does not have the CONTIGUOUS attribute or a pointer array.

#### NOTE 12.27

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.

#### 8 12.5.1.5 Co-array arguments

9 The actual argument shall be an allocatable co-array if and only if the dummy argument is an allocatable  
10 co-array with the same rank and co-rank.

11 If a dummy argument is a nonallocatable co-array, the co-rank and co-bounds are specified by the local  
12 declaration and are independent of those of the actual argument. The actual argument shall be a co-array  
13 or a subobject of a co-array without any co-subscripts, vector-valued subscripts, allocatable component  
14 selection, or pointer component selection.

**J3 internal note (cont.)****J3 internal note**

Unresolved Technical Issue 54

Broken. I assume you want to be able to pass

`VARIABLE%CO_ARRAY_COMPONENT`

to a dummy co-array, no?

Subgroup says yes. It's not entirely trivial to fix the wording though. (It probably needs to be something like "no *part-ref* after the *part-ref* with co-rank non-zero shall have the pointer or allocatable attribute, or have a vector-valued subscript".)

**NOTE 12.28**

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 12.29**

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 an assumed-shape co-array that does not have the CONTIGUOUS attribute or a subobject of an
- 2 assumed-shape co-array that does not have the CONTIGUOUS attribute is an actual argument associ-
- 3 ated with a dummy co-array, the dummy co-array shall be of assumed shape.
- 4 If an array section is an actual argument associated with a dummy co-array that is not of assumed shape
- 5 or has the CONTIGUOUS attribute, the section shall be contiguous (5.3.6).

**NOTE 12.30**

The constraints on actual arguments that correspond 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.

**J3 internal note**

Unresolved Technical Issue 79

This constraint on dummy co-arrays and their arguments appears to be unnecessary without some relaxation in the rules in 12.5.1.10 (the "high performance" rules for Fortran dummy arguments).

#### 1 12.5.1.6 Actual arguments associated with dummy procedure entities

2 If the actual argument is the name of an internal subprogram, the host instance of the dummy argument  
3 is the innermost currently executing instance of the host of that internal subprogram. If the actual  
4 argument has a host instance the host instance of the dummy argument is that instance. Otherwise the  
5 dummy argument has no host instance.

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

8 If a dummy argument is a dummy procedure without the POINTER attribute, the associated actual  
9 argument shall be an external, internal, module, or dummy procedure, or a specific intrinsic procedure  
10 listed in 13.6 and not marked with a bullet (●). If the specific name is also a generic name, only the  
11 specific procedure is associated with the dummy argument.

12 If an external procedure name or a dummy procedure name is used as an actual argument, its interface  
13 shall be explicit or it shall be explicitly declared to have the EXTERNAL attribute.

14 If the interface of the dummy argument is explicit, the characteristics listed in 12.3 shall be the same  
15 for the associated actual argument and the corresponding dummy argument, except that a pure actual  
16 argument may be associated with a dummy argument that is not pure and an elemental intrinsic actual  
17 procedure may be associated with a dummy procedure (which is prohibited from being elemental).

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

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

#### 23 12.5.1.7 Actual arguments associated with alternate return indicators

24 If a dummy argument is an asterisk (12.6.2.2), the associated actual argument shall be an alternate return specifier (12.5).

#### 25 12.5.1.8 Sequence association

26 An actual argument represents an **element sequence** if it is an array expression, an array element  
27 designator, a scalar of type default character, or a scalar of type character with the C character kind  
28 (15.2.2). If the actual argument is an array expression, the element sequence consists of the elements  
29 in array element order. If the actual argument is an array element designator, the element sequence  
30 consists of that array element and each element that follows it in array element order.

31 If the actual argument is of type default character or of type character with the C character kind, and is  
32 an array expression, array element, or array element substring designator, the element sequence consists  
33 of the storage units beginning with the first storage unit of the actual argument and continuing to the  
34 end of the array. The storage units of an array element substring designator are viewed as array elements  
35 consisting of consecutive groups of storage units having the character length of the dummy array.

36 If the actual argument is of type default character or of type character with the C character kind, and  
37 is a scalar that is not an array element or array element substring designator, the element sequence  
38 consists of the storage units of the actual argument.

#### NOTE 12.31

Some of the elements in the element sequence may consist of storage units from different elements of the original array.
--------------------------------------------------------------------------------------------------------------------------



1 An actual argument that represents an element sequence and corresponds to a dummy argument that is  
 2 an array is sequence associated with the dummy argument if the dummy argument is an explicit-shape  
 3 or assumed-size array. The rank and shape of the actual argument need not agree with the rank and  
 4 shape of the dummy argument, but the number of elements in the dummy argument shall not exceed  
 5 the number of elements in the element sequence of the actual argument. If the dummy argument is  
 6 assumed-size, the number of elements in the dummy argument is exactly the number of elements in the  
 7 element sequence.

#### 8 **12.5.1.9 Argument presence and restrictions on arguments not present**

9 A dummy argument or an entity that is host associated with a dummy argument is not **present** if the  
 10 dummy argument

- 11 (1) does not correspond to an actual argument,
- 12 (2) corresponds to an actual argument that is not present, or
- 13 (3) does not have the ALLOCATABLE or POINTER attribute, and corresponds to an actual  
 14 argument that
  - 15 (a) has the ALLOCATABLE attribute and is not allocated, or
  - 16 (b) has the POINTER attribute and is disassociated.

17 Otherwise, it is present. A nonoptional dummy argument shall be present. If an optional nonpointer  
 18 dummy argument corresponds to a pointer actual argument, the pointer association status of the actual  
 19 argument shall not be undefined.

20 An optional dummy argument that is not present is subject to the following restrictions.

- 21 (1) If it is a data object, it shall not be referenced or be defined. If it is of a type for which  
 22 default initialization is specified for some component, the initialization has no effect.
- 23 (2) It shall not be used as the *data-target* or *proc-target* of a pointer assignment.
- 24 (3) If it is a procedure or procedure pointer, it shall not be invoked.
- 25 (4) It shall not be supplied as an actual argument corresponding to a nonoptional dummy  
 26 argument other than as the argument of the PRESENT intrinsic function or as an argument  
 27 of a function reference that meets the requirements of (6) or (8) in 7.1.7.
- 28 (5) A designator with it as the base object and with at least one subobject selector shall not  
 29 be supplied as an actual argument.
- 30 (6) If it is an array, it shall not be supplied as an actual argument to an elemental procedure  
 31 unless an array of the same rank is supplied as an actual argument corresponding to a  
 32 nonoptional dummy argument of that elemental procedure.
- 33 (7) If it is a pointer, it shall not be allocated, deallocated, nullified, pointer-assigned, or supplied  
 34 as an actual argument corresponding to an optional nonpointer dummy argument.
- 35 (8) If it is allocatable, it shall not be allocated, deallocated, or supplied as an actual argument  
 36 corresponding to an optional nonallocatable dummy argument.
- 37 (9) If it has length type parameters, they shall not be the subject of an inquiry.
- 38 (10) It shall not be used as the *selector* in a SELECT TYPE or ASSOCIATE construct.

39 Except as noted in the list above, it may be supplied as an actual argument corresponding to an optional  
 40 dummy argument, which is then also considered not to be associated with an actual argument.

#### 41 **12.5.1.10 Restrictions on entities associated with dummy arguments**

42 While an entity is associated with a dummy argument, the following restrictions hold.

- 43 (1) Action that affects the allocation status of the entity or a subobject thereof shall be taken  
 44 through the dummy argument. Action that affects the value of the entity or any subobject

- 1 of it shall be taken only through the dummy argument unless
- 2 (a) the dummy argument has the POINTER attribute or
- 3 (b) the dummy argument has the TARGET attribute, the dummy argument does not
- 4 have INTENT (IN), the dummy argument is a scalar object or an assumed-shape
- 5 array, and the actual argument is a target other than an array section with a vector
- 6 subscript.

**NOTE 12.32**

In

```

SUBROUTINE OUTER
  REAL, POINTER :: A (:)
  ...
  ALLOCATE (A (1:N))
  ...
  CALL INNER (A)
  ...
CONTAINS
  SUBROUTINE INNER (B)
    REAL :: B (:)
    ...
  END SUBROUTINE INNER
  SUBROUTINE SET (C, D)
    REAL, INTENT (OUT) :: C
    REAL, INTENT (IN) :: D
    C = D
  END SUBROUTINE SET
END SUBROUTINE OUTER

```

an assignment statement such as

```
A (1) = 1.0
```

would not be permitted during the execution of INNER because this would be changing A without using B, but statements such as

```
B (1) = 1.0
```

or

```
CALL SET (B (1), 1.0)
```

would be allowed. Similarly,

```
DEALLOCATE (A)
```

would not be allowed because this affects the allocation of B without using B. In this case,

```
DEALLOCATE (B)
```

also would not be permitted. If B were declared with the POINTER attribute, either of the statements

**NOTE 12.32 (cont.)**

DEALLOCATE (A)

and

DEALLOCATE (B)

would be permitted, but not both.

**NOTE 12.33**

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 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 12.34**

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 12.35**

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
- 3 the allocation or deallocation, it may be referenced only through the dummy argument. If
- 4 the value the entity or any subobject of it is affected through the dummy argument, then
- 5 at any time during the execution of the procedure, either before or after the definition, it
- 6 may 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

- 1 array, and the actual argument is a target other than an array section with a vector  
 2 subscript.

**NOTE 12.36**

In

```

MODULE DATA
  REAL :: W, X, Y, Z
END MODULE DATA

PROGRAM MAIN
  USE DATA
  ...
  CALL INIT (X)
  ...
END PROGRAM MAIN
SUBROUTINE INIT (V)
  USE DATA
  ...
  READ (*, *) V
  ...
END SUBROUTINE INIT

```

variable X 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 12.37**

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.

### 3 12.5.2 Function reference

- 4 A function is invoked during expression evaluation by a *function-reference* or by a defined operation  
 5 (7.1.3). When it is invoked, all actual argument expressions are evaluated, then the arguments are  
 6 associated, and then the function is executed. When execution of the function is complete, the value  
 7 of the function result is available for use in the expression that caused the function to be invoked. The  
 8 characteristics of the function result (12.3.2) are determined by the interface of the function. A reference  
 9 to an elemental function (12.8) is an elemental reference if one or more actual arguments are arrays and  
 10 all array arguments have the same shape.

### 11 12.5.3 Subroutine reference

- 12 A subroutine is invoked by execution of a CALL statement, execution of a defined assignment statement  
 13 (7.4.1.4), user-defined derived-type input/output(9.5.3.7.1), or finalization(4.5.6). When a subroutine is  
 14 invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the  
 15 subroutine is executed. When the actions specified by the subroutine are completed, the execution of  
 16 the CALL statement, the execution of the defined assignment statement, or the processing of an input  
 17 or output list item is also completed. If a CALL statement includes one or more alternate return specifiers among  
 18 its arguments, control may be transferred to one of the statements indicated, depending on the action specified by the

1 subroutine. A reference to an elemental subroutine (12.8) is an elemental reference if there is at least one  
 2 actual argument corresponding to an INTENT(OUT) or INTENT(INOUT) dummy argument, all such  
 3 actual arguments are arrays, and all actual arguments are conformable.

#### 4 **12.5.4 Resolving named procedure references**

5 The rules for interpreting a procedure reference depend on whether the procedure name in the reference  
 6 is established by the available declarations and specifications to be generic in the scoping unit containing  
 7 the reference, is established to be only specific in the scoping unit containing the reference, or is not  
 8 established.

9 A procedure name is established to be generic in a scoping unit

- 10 (1) if that scoping unit contains an interface block with that name;
- 11 (2) if that scoping unit contains an INTRINSIC attribute specification for that name and it is  
 12 the name of a generic intrinsic procedure;
- 13 (3) if that scoping unit contains a USE statement that makes that procedure name accessible  
 14 and the corresponding name in the module is established to be generic; or
- 15 (4) if that scoping unit contains no declarations of that name, that scoping unit has a host  
 16 scoping unit, and that name is established to be generic in the host scoping unit.

17 A procedure name is established to be only specific in a scoping unit if it is established to be specific  
 18 and not established to be generic. It is established to be specific

- 19 (1) if that scoping unit contains a module subprogram, internal subprogram, or statement function  
 20 that defines a procedure with that name;
- 21 (2) if that scoping unit contains an INTRINSIC attribute specification for that name and if it  
 22 is the name of a specific intrinsic procedure;
- 23 (3) if that scoping unit contains an explicit EXTERNAL attribute specification (5.3.8) for that  
 24 name;
- 25 (4) if that scoping unit contains a USE statement that makes that procedure name accessible  
 26 and the corresponding name in the module is established to be specific; or
- 27 (5) if that scoping unit contains no declarations of that name, that scoping unit has a host  
 28 scoping unit, and that name is established to be specific in the host scoping unit.

29 A procedure name is not established in a scoping unit if it is neither established to be generic nor  
 30 established to be specific.

##### 31 **12.5.4.1 Resolving procedure references to names established to be generic**

32 If the reference is consistent with a nonelemental reference to one of the specific interfaces of a generic  
 33 interface that has that name and either is in the scoping unit in which the reference appears or is made  
 34 accessible by a USE statement in the scoping unit, the reference is to the specific procedure in the  
 35 interface block that provides that interface. The rules in 16.3.4 ensure that there can be at most one  
 36 such specific procedure.

37 Otherwise, if the reference is consistent with an elemental reference to one of the specific interfaces of  
 38 a generic interface that has that name and either is in the scoping unit in which the reference appears  
 39 or is made accessible by a USE statement in the scoping unit, the reference is to the specific elemental  
 40 procedure in the interface block that provides that interface. The rules in 16.3.4 ensure that there can  
 41 be at most one such specific elemental procedure.

42 Otherwise, if the scoping unit contains either an INTRINSIC attribute specification for that name or  
 43 a USE statement that makes that name accessible from a module in which the corresponding name is  
 44 specified to have the INTRINSIC attribute, and if the reference is consistent with the interface of that  
 45 intrinsic procedure, the reference is to that intrinsic procedure.

- 1 Otherwise, if the scoping unit has a host scoping unit, if the name is established to be generic in that
- 2 host scoping unit, and if there is agreement between the scoping unit and the host scoping unit as to
- 3 whether the name is a function name or a subroutine name, the name is resolved by applying the rules
- 4 in this subclause to the host scoping unit.

**NOTE 12.38**

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 = RANF(AA(6:10,2))
```

is a nonelemental reference to VECTOR\_RANDOM.

**NOTE 12.39**

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 12.5.4.2 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
- 16 intrinsic with that name.

- 1 If the scoping unit contains a USE statement that makes a procedure accessible by the name, the  
2 reference is to that procedure.

**NOTE 12.40**

Because of the renaming facility of the USE statement, the name in the reference may be different from the original name of the procedure.

- 3 If none of the above apply, the scoping unit shall have a host scoping unit, and the reference is resolved  
4 by applying the rules in this subclause to the host scoping unit.

**5 12.5.4.3 Resolving procedure references to names not established**

- 6 If the name is the name of a dummy argument of the scoping unit, the dummy argument is a dummy  
7 procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing  
8 that reference is the procedure supplied as the actual argument corresponding to that dummy procedure.

- 9 Otherwise, if the name is the name of an intrinsic procedure, and if there is agreement between the  
10 reference and the status of the intrinsic procedure as being a function or subroutine, the reference is to  
11 that intrinsic procedure.

- 12 Otherwise, the reference is to an external procedure with that name.

**13 12.5.5 Resolving type-bound procedure references**

- 14 If the *binding-name* in a *procedure-designator* (R1221) is that of a specific type-bound procedure, the  
15 procedure referenced is the one bound to that name in the dynamic type of the *data-ref*.

- 16 If the *binding-name* in a *procedure-designator* is that of a generic type bound procedure, the generic  
17 binding with that name in the declared type of the *data-ref* is used to select a specific binding using the  
18 following criteria.

- 19 (1) If the reference is consistent with one of the specific bindings of that generic binding, that  
20 specific binding is selected.  
21 (2) Otherwise, the reference shall be consistent with an elemental reference to one of the specific  
22 bindings of that generic binding; that specific binding is selected.

- 23 The reference is to the procedure bound to the same name as the selected specific binding in the dynamic  
24 type of the *data-ref*.

**25 12.6 Procedure definition****26 12.6.1 Intrinsic procedure definition**

- 27 Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor  
28 shall include the intrinsic procedures described in Clause 13, but may include others. However, a  
29 standard-conforming program shall not make use of intrinsic procedures other than those described in  
30 Clause 13.

**31 12.6.2 Procedures defined by subprograms**

- 32 When a procedure defined by a subprogram is invoked, an instance (12.6.2.3) of the subprogram is  
33 created and executed. Execution begins with the first executable construct following the FUNCTION,  
34 SUBROUTINE, or ENTRY statement specifying the name of the procedure invoked.

### 1 12.6.2.1 Function subprogram

2 A **function subprogram** is a subprogram that has a FUNCTION statement as its first statement.

```

3 R1225 function-subprogram      is function-stmt
4                                [ specification-part ]
5                                [ execution-part ]
6                                [ internal-subprogram-part ]
7                                end-function-stmt
8 R1226 function-stmt           is [ prefix ] FUNCTION function-name ■
9                                ■ ( [ dummy-arg-name-list ] ) [ suffix ]
```

10 C1238 (R1226) If RESULT appears, *result-name* shall not be the same as *function-name* and shall not  
 11 be the same as the *entry-name* in any ENTRY statement in the subprogram.

12 C1239 (R1226) If RESULT appears, the *function-name* shall not appear in any specification statement  
 13 in the scoping unit of the function subprogram.

```

14 R1227 proc-language-binding-spec is language-binding-spec
```

15 C1240 (R1227) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the  
 16 *function-stmt* or *subroutine-stmt* of an interface body for an abstract interface or a dummy  
 17 procedure.

18 C1241 (R1227) A *proc-language-binding-spec* shall not be specified for an internal procedure.

19 C1242 (R1227) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy  
 20 arguments shall be a nonoptional interoperable variable (15.3.5, 15.3.6) or a nonoptional interoper-  
 21 able procedure (15.3.7). If *proc-language-binding-spec* is specified for a function, the function  
 22 result shall be an interoperable scalar variable.

```

23 R1228 dummy-arg-name          is name
```

24 C1243 (R1228) A *dummy-arg-name* shall be the name of a dummy argument.

```

25 R1229 prefix                  is prefix-spec [ prefix-spec ] ...
26 R1230 prefix-spec             is declaration-type-spec
27                                or ELEMENTAL
28                                or IMPURE
29                                or MODULE
30                                or PURE
31                                or RECURSIVE
```

32 C1244 (R1229) A *prefix* shall contain at most one of each *prefix-spec*.

33 C1245 (R1229) A *prefix* shall not specify both PURE and IMPURE.

34 C1246 (R1229) A *prefix* shall not specify both ELEMENTAL and RECURSIVE.

35 C1247 (R1229) A *prefix* shall not specify ELEMENTAL if *proc-language-binding-spec* appears in the  
 36 *function-stmt* or *subroutine-stmt*.

37 C1248 (R1229) MODULE shall appear only within the *function-stmt* or *subroutine-stmt* of a module  
 38 subprogram or of an interface body that is declared in the scoping unit of a module or submodule.

39 C1249 (R1229) If MODULE appears within the *prefix* in a module subprogram, an accessible module  
 40 procedure interface having the same name as the subprogram shall be declared in the module  
 41 or submodule in which the subprogram is defined, or shall be declared in an ancestor of that



- 1           program unit.
- 2 C1250 (R1229) If MODULE appears within the *prefix* in a module subprogram, the subprogram shall  
3 specify the same characteristics and dummy argument names as its corresponding (12.6.2.4)  
4 module procedure interface body.
- 5 C1251 (R1229) If MODULE appears within the *prefix* in a module subprogram and a binding label  
6 is specified, it shall be the same as the binding label specified in the corresponding module  
7 procedure interface body.
- 8 C1252 (R1229) If MODULE appears within the *prefix* in a module subprogram, RECURSIVE shall  
9 appear if and only if RECURSIVE appears in the *prefix* in the corresponding module procedure  
10 interface body.
- 11 R1231 *suffix*                                   is *proc-language-binding-spec* [ RESULT ( *result-name* ) ]  
12                                                   or RESULT ( *result-name* ) [ *proc-language-binding-spec* ]
- 13 R1232 *end-function-stmt*                   is END [ FUNCTION [ *function-name* ] ]
- 14 C1253 (R1225) An internal function subprogram shall not contain an ENTRY statement.
- 15 C1254 (R1225) An internal function subprogram shall not contain an *internal-subprogram-part*.
- 16 C1255 (R1232) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-*  
17 *name* specified in the *function-stmt*.
- 18 The name of the function is *function-name*.
- 19 The type and type parameters (if any) of the result of the function defined by a function subprogram  
20 may be specified by a type specification in the FUNCTION statement or by the name of the result  
21 variable appearing in a type declaration statement in the declaration part of the function subprogram.  
22 They shall not be specified both ways. If they are not specified either way, they are determined by  
23 the implicit typing rules in force within the function subprogram. If the function result is an array,  
24 allocatable, or a pointer, this shall be specified by specifications of the name of the result variable within  
25 the function body. The specifications of the function result attributes, the specification of dummy  
26 argument attributes, and the information in the procedure heading collectively define the characteristics  
27 of the function (12.3).
- 28 The *prefix-spec* RECURSIVE shall appear if the function directly or indirectly invokes itself or a function  
29 defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall appear if a  
30 function defined by an ENTRY statement in the subprogram directly or indirectly invokes itself, another  
31 function defined by an ENTRY statement in that subprogram, or the function defined by the FUNCTION  
32 statement.
- 33 If RESULT appears, the name of the result variable of the function is *result-name* and all occurrences  
34 of the function name in *execution-part* statements in the scoping unit refer to the function itself. If  
35 RESULT does not appear, the result variable is *function-name* and all occurrences of the function name  
36 in *execution-part* statements in the scoping unit are references to the result variable. The characteristics  
37 (12.3.2) of the function result are those of the result variable. On completion of execution of the function,  
38 the value returned is that of its result variable. If the function result is a pointer, the shape of the value  
39 returned by the function is determined by the shape of the result variable when the execution of the  
40 function is completed. If the result variable is not a pointer, its value shall be defined by the function.  
41 If the function result is a pointer, the function shall either associate a target with the result variable  
42 pointer or cause the association status of this pointer to become defined as disassociated.

**NOTE 12.41**

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.

- 1 If the *prefix-spec* PURE appears, or the *prefix-spec* ELEMENTAL appears and IMPURE does not appear,
- 2 the subprogram is a pure subprogram and shall meet the additional constraints of 12.7.
- 3 If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet
- 4 the additional constraints of 12.8.1.

**NOTE 12.42**

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
    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 12.43**

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 12.6.2.2 Subroutine subprogram

2 A **subroutine subprogram** is a subprogram that has a SUBROUTINE statement as its first statement.

```

3 R1233 subroutine-subprogram      is subroutine-stmt
4                                   [ specification-part ]
5                                   [ execution-part ]
6                                   [ internal-subprogram-part ]
7                                   end-subroutine-stmt
8 R1234 subroutine-stmt             is [ prefix ] SUBROUTINE subroutine-name ■
9                                   ■ [ ( [ dummy-arg-list ] ) [ proc-language-binding-spec ] ]
```

10 C1256 (R1234) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

```

11 R1235 dummy-arg                  is dummy-arg-name
12                                or *
```

```

13 R1236 end-subroutine-stmt        is END [ SUBROUTINE [ subroutine-name ] ]
```

14 C1257 (R1233) An internal subroutine subprogram shall not contain an ENTRY statement.

15 C1258 (R1233) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

16 C1259 (R1236) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the  
17 *subroutine-name* specified in the *subroutine-stmt*.

18 The name of the subroutine is *subroutine-name*.

19 The *prefix-spec* RECURSIVE shall appear if the subroutine directly or indirectly invokes itself or a  
20 subroutine defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall  
21 appear if a subroutine defined by an ENTRY statement in the subprogram directly or indirectly invokes  
22 itself, another subroutine defined by an ENTRY statement in that subprogram, or the subroutine defined  
23 by the SUBROUTINE statement.

24 If the *prefix-spec* PURE appears, or the *prefix-spec* ELEMENTAL appears and IMPURE does not appear,  
25 the subprogram is a pure subprogram and shall meet the additional constraints of 12.7.

26 If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet  
27 the additional constraints of 12.8.1.

### 28 12.6.2.3 Instances of a subprogram

29 When a function or subroutine defined by a subprogram is invoked, an **instance** of that subprogram is  
30 created. When a statement function is invoked, an instance of that statement function is created.

31 Each instance has an independent sequence of execution and an independent set of dummy arguments  
32 and local unsaved data objects. If an internal procedure or statement function in the subprogram is invoked  
33 by name from an instance of the subprogram or from an internal subprogram or statement function that  
34 has access to the entities of that instance, the created instance of the internal subprogram or statement  
35 function also has access to the entities of that instance of the host subprogram. If an internal procedure  
36 is invoked via a dummy procedure or procedure pointer, the internal procedure has access to the entities  
37 of the host instance of that dummy procedure or procedure pointer.

38 All other entities are shared by all instances of the subprogram.

**NOTE 12.44**

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.

**1 12.6.2.4 Separate module procedures**

2 A **separate module procedure** is a module procedure defined by a *separate-module-subprogram*,  
 3 by a *function-subprogram* whose initial statement contains the keyword MODULE, or by a *subroutine-*  
 4 *subprogram* whose initial statement contains the keyword MODULE. Its interface is declared by a module  
 5 procedure interface body (12.4.2.1) in the *specification-part* of the module or submodule in which the  
 6 procedure is defined, or in an ancestor module or submodule.

7 R1237 *separate-module-subprogram* **is** MODULE PROCEDURE *procedure-name*  
 8 [ *specification-part* ]  
 9 [ *execution-part* ]  
 10 [ *internal-subprogram-part* ]  
 11 *end-sep-subprogram-stmt*

12 R1238 *end-sep-subprogram-stmt* **is** END [PROCEDURE [*procedure-name*]]

13 C1260 (R1237) The *procedure-name* shall be the same as the name of an accessible module procedure  
 14 interface that is declared in the module or submodule in which the *separate-module-subprogram*  
 15 is defined, or is declared in an ancestor of that program unit.

16 C1261 (R1238) If a *procedure-name* appears in the *end-sep-subprogram-stmt*, it shall be identical to the  
 17 *procedure-name* in the MODULE PROCEDURE statement.

18 A module procedure interface body and a subprogram that defines a separate module procedure **corre-**  
 19 **spond** if they have the same name, and the module procedure interface is declared in the same program  
 20 unit as the subprogram or is declared in an ancestor of the program unit in which the procedure is  
 21 defined and is accessible by host association from that ancestor. A module procedure interface body  
 22 shall not correspond to more than one subprogram that defines a separate module procedure.

**NOTE 12.45**

A separate module procedure is only accessible by use association if its interface body is declared in the specification part of a module and is public.

23 If a procedure is defined by a *separate-module-subprogram*, its characteristics are specified by the corre-  
 24 sponding module procedure interface body.

25 If a separate module procedure is a function defined by a *separate-module-subprogram*, the result variable  
 26 name is determined by the FUNCTION statement in the module procedure interface body. Otherwise,  
 27 the result variable name is determined by the FUNCTION statement in the module subprogram.

**28 12.6.2.5 ENTRY statement**

29 An **ENTRY statement** permits a procedure reference to begin with a particular executable statement  
 30 within the function or subroutine subprogram in which the ENTRY statement appears.

31 R1239 *entry-stmt* **is** ENTRY *entry-name* [ ( [ *dummy-arg-list* ] ) [ *suffix* ] ]

32 C1262 (R1239) If RESULT appears, the *entry-name* shall not appear in any specification or type-  
 33 declaration statement in the scoping unit of the function program.

34 C1263 (R1239) An *entry-stmt* shall appear only in an *external-subprogram* or a *module-subprogram*

1           that does not define a separate module procedure. An *entry-stmt* shall not appear within an  
2           *executable-construct*.

3 C1264 (R1239) RESULT shall appear only if the *entry-stmt* is in a function subprogram.

4 C1265 (R1239) Within the subprogram containing the *entry-stmt*, the *entry-name* shall not appear  
5           as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY  
6           statement nor shall it appear in an EXTERNAL, INTRINSIC, or PROCEDURE statement.

7 C1266 (R1239) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function  
8           subprogram.

9 C1267 (R1239) If RESULT appears, *result-name* shall not be the same as the *function-name* in the  
10          FUNCTION statement and shall not be the same as the *entry-name* in any ENTRY statement  
11          in the subprogram.

12   Optionally, a subprogram may have one or more ENTRY statements.

13   If the ENTRY statement is in a function subprogram, an additional function is defined by that subpro-  
14   gram. The name of the function is *entry-name* and the name of its result variable is *result-name* or  
15   is *entry-name* if no *result-name* is provided. The characteristics of the function result are specified by  
16   specifications of the result variable. The dummy arguments of the function are those specified in the  
17   ENTRY statement. If the characteristics of the result of the function named in the ENTRY statement  
18   are the same as the characteristics of the result of the function named in the FUNCTION statement,  
19   their result variables identify the same variable, although their names need not be the same. Otherwise,  
20   they are storage associated and shall all be nonpointer, nonallocatable scalars of type default integer,  
21   default real, double precision real, default complex, or default logical.

22   If the ENTRY statement is in a subroutine subprogram, an additional subroutine is defined by that  
23   subprogram. The name of the subroutine is *entry-name*. The dummy arguments of the subroutine are  
24   those specified in the ENTRY statement.

25   The order, number, types, kind type parameters, and names of the dummy arguments in an ENTRY  
26   statement may differ from the order, number, types, kind type parameters, and names of the dummy  
27   arguments in the FUNCTION or SUBROUTINE statement in the containing subprogram.

28   Because an ENTRY statement defines an additional function or an additional subroutine, it is referenced  
29   in the same manner as any other function or subroutine (12.5).

30   In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear  
31   in an executable statement preceding that ENTRY statement, unless it also appears in a FUNCTION,  
32   SUBROUTINE, or ENTRY statement that precedes the executable statement.

33   In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear in the expression  
34   of a statement function unless the name is also a dummy argument of the statement function, appears in a FUNCTION  
35   or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

36   If a dummy argument appears in an executable statement, the execution of the executable statement is  
37   permitted during the execution of a reference to the function or subroutine only if the dummy argument  
38   appears in the dummy argument list of the procedure name referenced.

39   If a dummy argument is used in a specification expression to specify an array bound or character length  
40   of an object, the appearance of the object in a statement that is executed during a procedure reference  
41   is permitted only if the dummy argument appears in the dummy argument list of the procedure name  
42   referenced and it is present (12.5.1.9).

43   A scoping unit containing a reference to a procedure defined by an ENTRY statement may have access to

1 an interface body for the procedure. The procedure header for the interface body shall be a FUNCTION  
 2 statement for an entry in a function subprogram and shall be a SUBROUTINE statement for an entry  
 3 in a subroutine subprogram.

4 The keyword RECURSIVE is not used in an ENTRY statement. Instead, the presence or absence of  
 5 RECURSIVE in the initial SUBROUTINE or FUNCTION statement controls whether the procedure  
 6 defined by an ENTRY statement is permitted to reference itself.

7 The keywords PURE and IMPURE are not used in an ENTRY statement. Instead, the procedure  
 8 defined by an ENTRY statement is pure if and only if PURE is specified or ELEMENTAL is specified  
 9 and IMPURE is not specified in the SUBROUTINE or FUNCTION statement.

10 The keyword ELEMENTAL is not used in an ENTRY statement. Instead, the procedure defined by  
 11 an ENTRY statement is elemental if and only if ELEMENTAL is specified in the SUBROUTINE or  
 12 FUNCTION statement.

### 13 12.6.2.6 RETURN statement

14 R1240 *return-stmt* is RETURN [ *scalar-int-expr* ]

15 C1268 (R1240) The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.

16 C1269 (R1240) The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

17 Execution of the RETURN statement completes execution of the instance of the subprogram in which  
 18 it appears. If the expression appears and has a value *n* between 1 and the number of asterisks in the dummy argument  
 19 list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate  
 20 return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is  
 21 no transfer of control to an alternate return.

22 Execution of an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a RETURN state-  
 23 ment with no expression.

### 24 12.6.2.7 CONTAINS statement

25 R1241 *contains-stmt* is CONTAINS

26 The CONTAINS statement separates the body of a main program, module, submodule, or subprogram  
 27 from any internal or module subprograms it may contain, or it introduces the type-bound procedure  
 28 part of a derived-type definition (4.5.2). The CONTAINS statement is not executable.

## 29 12.6.3 Definition and invocation of procedures by means other than Fortran

30 A procedure may be defined by means other than Fortran. The interface of a procedure defined by  
 31 means other than Fortran may be specified by an interface body or procedure declaration statement. If  
 32 the interface of such a procedure does not have a *proc-language-binding-spec*, the means by which the  
 33 procedure is defined are processor dependent. A reference to such a procedure is made as though it were  
 34 defined by an external subprogram.

35 If the interface of a procedure has a *proc-language-binding-spec*, the procedure is interoperable (15.5).

36 Interoperation with C functions is described in 15.5.

#### NOTE 12.46

For explanatory information on definition of procedures by means other than Fortran, see subclause C.9.2.

## 1 12.6.4 Statement function

2 A statement function is a function defined by a single statement.

3 R1242 *stmt-function-stmt* **is** *function-name* ( [ *dummy-arg-name-list* ] ) = *scalar-expr*

4 C1270 (R1242) The *primaries* of the *scalar-expr* shall be constants (literal and named), references to variables, references  
5 to functions and function dummy procedures, and intrinsic operations. If *scalar-expr* contains a reference to a  
6 function or a function dummy procedure, the reference shall not require an explicit interface, the function shall  
7 not require an explicit interface unless it is an intrinsic, the function shall not be a transformational intrinsic,  
8 and the result shall be scalar. If an argument to a function or a function dummy procedure is an array, it shall  
9 be an array name. If a reference to a statement function appears in *scalar-expr*, its definition shall have been  
10 provided earlier in the scoping unit and shall not be the name of the statement function being defined.

11 C1271 (R1242) Named constants in *scalar-expr* shall have been declared earlier in the scoping unit or made accessible  
12 by use or host association. If array elements appear in *scalar-expr*, the array shall have been declared as an array  
13 earlier in the scoping unit or made accessible by use or host association.

14 C1272 (R1242) If a *dummy-arg-name*, variable, function reference, or dummy function reference is typed by the implicit  
15 typing rules, its appearance in any subsequent type declaration statement shall confirm this implied type and  
16 the values of any implied type parameters.

17 C1273 (R1242) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.

18 C1274 (R1242) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.

19 C1275 (R1242) Each variable reference in *scalar-expr* may be either a reference to a dummy argument of the statement  
20 function or a reference to a variable accessible in the same scoping unit as the statement function statement.

21 The definition of a statement function with the same name as an accessible entity from the host shall be preceded by the  
22 declaration of its type in a type declaration statement.

23 The dummy arguments have a scope of the statement function statement. Each dummy argument has the same type and  
24 type parameters as the entity of the same name in the scoping unit containing the statement function.

25 A statement function shall not be supplied as a procedure argument.

26 The value of a statement function reference is obtained by evaluating the expression using the values of the actual arguments  
27 for the values of the corresponding dummy arguments and, if necessary, converting the result to the declared type and  
28 type attributes of the function.

29 A function reference in the scalar expression shall not cause a dummy argument of the statement function to become  
30 redefined or undefined.

## 31 12.7 Pure procedures

32 A pure procedure is

- 33 (1) a pure intrinsic function (13.1),
- 34 (2) a pure intrinsic subroutine (13.1),
- 35 (3) defined by a pure subprogram, or
- 36 (4) a statement function that references only pure functions.

37 A pure subprogram is a subprogram that has the *prefix-spec* PURE or that has the *prefix-spec* ELE-  
38 MENTAL and does not have the *prefix-spec* IMPURE. The following additional constraints apply to

- 1 pure subprograms.
- 2 C1276 The *specification-part* of a pure function subprogram shall specify that all its nonpointer dummy  
3 data objects have INTENT(IN).
- 4 C1277 The *specification-part* of a pure subroutine subprogram shall specify the intents of all its non-  
5 pointer dummy data objects.
- 6 C1278 A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure subpro-  
7 gram shall not have the SAVE attribute.

**NOTE 12.47**

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; therefore, such initialization is also disallowed.

- 8 C1279 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are  
9 pure.
- 10 C1280 If a procedure that is neither an intrinsic procedure nor a statement function is used in a context  
11 that requires it to be pure, then its interface shall be explicit in the scope of that use. The  
12 interface shall specify that the procedure is pure.
- 13 C1281 All internal subprograms in a pure subprogram shall be pure.
- 14 C1282 In a pure subprogram any designator with a base object that is in common or accessed by  
15 host or use association, is a dummy argument of a pure function, is a dummy argument with  
16 INTENT (IN) of a pure subroutine, or an object that is storage associated with any such variable,  
17 shall not be used
- 18 (1) in a variable definition context(16.6.7),  
19 (2) as the *data-target* in a *pointer-assignment-stmt*,  
20 (3) as the *expr* corresponding to a component with the POINTER attribute in a *structure-*  
21 *constructor*,  
22 (4) as the *expr* of an intrinsic assignment statement in which the variable is of a derived type  
23 if the derived type has a pointer component at any level of component selection, or  
24 (5) as an actual argument associated with a dummy argument with INTENT (OUT) or IN-  
25 TENT (INOUT) or with the POINTER attribute.

**NOTE 12.48**

Item 3 requires that processors be able to determine if entities with the PRIVATE attribute or with private components have a pointer component.



- 1 C1283 Any procedure referenced in a pure subprogram, including one referenced via a defined operation,  
2 assignment, or finalization, shall be pure.
- 3 C1284 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-*  
4 *stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.
- 5 C1285 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-*  
6 *number* or *\**.
- 7 C1286 A pure subprogram shall not contain a *stop-stmt*.
- 8 C1287 A co-indexed object shall not appear in a variable definition context in a pure subprogram.
- 9 C1288 A pure subprogram shall not contain an image control statement (8.5.1).

**NOTE 12.49**

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 12.50**

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignment-stmts* to be defined assignments. The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to *INTENT (OUT)*, *INTENT (INOUT)*, and pointer dummy arguments are permitted.

## 10 12.8 Elemental procedures

### 11 12.8.1 Elemental procedure declaration and interface

12 An **elemental procedure** is an elemental intrinsic procedure or a procedure that is defined by an  
13 elemental subprogram.

14 An elemental subprogram has the *prefix-spec* *ELEMENTAL*. An elemental subprogram is a pure sub-  
15 program unless it has the *prefix-spec* *IMPURE*. The following additional constraints apply to elemental

- 1 subprograms.
- 2 C1289 All dummy arguments of an elemental procedure shall be scalar dummy data objects and shall  
3 not have the POINTER or ALLOCATABLE attribute.
- 4 C1290 The result variable of an elemental function shall be scalar and shall not have the POINTER or  
5 ALLOCATABLE attribute.
- 6 C1291 In the scoping unit of an elemental subprogram, an object designator with a dummy argument  
7 as the base object shall not appear in a *specification-expr* except as the argument to one of the  
8 intrinsic functions BIT\_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

**NOTE 12.51**

An elemental subprogram is a pure subprogram and all of the constraints for pure subprograms also apply.

**NOTE 12.52**

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

```

## 9 12.8.2 Elemental function actual arguments and results

- 10 If a generic name or a specific name is used to reference an elemental function, the shape of the result is  
11 the same as the shape of the actual argument with the greatest rank. If there are no actual arguments  
12 or the actual arguments are all scalar, the result is scalar. For those elemental functions that have more  
13 than one argument, all actual arguments shall be conformable. In the array case, the values of the  
14 elements, if any, of the result are the same as would have been obtained if the scalar function had been  
15 applied separately, in array element order, to corresponding elements of each array actual argument.

**NOTE 12.53**

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

**NOTE 12.53 (cont.)**

$\text{MAX}(X(I, J), 0.0, Y(I, J)), I = 1, 2, \dots, M, J = 1, 2, \dots, N$
-----------------------------------------------------------------------------

**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 12.5.1.10.



# 1 13 Intrinsic procedures and modules

## 2 13.1 Classes of intrinsic procedures

3 There are four classes of intrinsic procedures: inquiry functions, elemental functions, transformational  
4 functions, and subroutines. Some intrinsic subroutines are elemental. Some intrinsic subroutines are  
5 collective.

6 An **inquiry function** is one whose result depends on the properties of one or more of its arguments  
7 instead of their values; in fact, these argument values may be undefined. Unless the description of  
8 an inquiry function states otherwise, these arguments are permitted to be unallocated allocatables or  
9 pointers that are not associated. An **elemental intrinsic function** is one that is specified for scalar  
10 arguments, but may be applied to array arguments as described in 12.8. All other intrinsic functions  
11 are **transformational functions**; they almost all have one or more array arguments or an array result.  
12 All standard intrinsic functions are pure.

13 The subroutine MOVE\_ALLOC and the elemental subroutine MVBITS are pure. No other standard  
14 intrinsic subroutine is pure.

15 A **collective subroutine** is one that has an argument of type IMAGE\_TEAM (13.8.3.5). If it is invoked  
16 by one image of a team, it shall be invoked by the same statement on all images of the team. There  
17 is an implicit team synchronization (8.5.3) at the beginning and end of the execution of a collective  
18 subroutine.

### J3 internal note

#### Unresolved Technical Issue 56

According to the above definition, if a user-defined procedure has a dummy argument of type IMAGE\_TEAM, it is a collective subroutine *AND THEREFORE* there is an implicit team synchronisation at the beginning and end of its execution.

You cannot be serious.

Also, RESHAPE, TRANSFER, and SPREAD can all take arguments of type IMAGE\_TEAM, so that makes them collective subroutines when given such an argument?

### J3 internal note

#### Unresolved Technical Issue 57

The term “implicit team synchronization” is used twice and never defined.

*Define it.*

It is not as simple as “all the members of the team”: which team – an image can be a member of several teams, no? Similarly, it cannot be as simple as “the value of the argument type IMAGE\_TEAM, since that is an output argument of FORM\_TEAM.

### NOTE 13.1

The subroutine FORM\_TEAM forms a team of images. The other collective subroutines return an array of the same shape as the given co-array after having applied an operation on the images in a team. The simple case of wanting the operation to be applied to all the elements of the co-array can easily be programmed, for example:

```
REAL FUNCTION GLOBAL_SUM(ARRAY)
  REAL      :: ARRAY(:, :)[*]
  REAL,SAVE :: TEMP[*]
```

**NOTE 13.1 (cont.)**

```

TEMP = SUM(ARRAY) ! Sum on the executing image
CALL CO_SUM(TEMP,GLOBAL_SUM)
END FUNCTION GLOBAL_SUM

```

Versions with other arguments can be programmed similarly, see C.10.2.

**J3 internal note****Unresolved Technical Issue 58**

Sorry, but this tutorial does not belong in “Classes of intrinsic procedures”. Even the normative stuff is probably a bit long for that (if there were a subclause about collective subroutines this bit should be simplified and the details moved there).

Since this is all basic “how to do parallel programming with coarrays”, I’m not sure it belongs in the standard at all; if it is needed, it ought to be in a separate subclause.

Why the advertising-speak “easily be programmed”? That is inappropriate for a standard.

Finally, this invocation of CO\_SUM does not have any argument of type IMAGE\_TEAM, and therefore is not a collective subroutine!

**NOTE 13.2**

As with user-written elemental subroutines, an elemental intrinsic subroutine is pure. The effects of MOVE\_ALLOC are limited to its arguments. The remaining nonelemental intrinsic subroutines all have side effects (or reflect system side effects) and thus are not pure.

1 **Generic names** of standard intrinsic procedures are listed in 13.5. In most cases, generic functions  
 2 accept arguments of more than one type and the type of the result is the same as the type of the  
 3 arguments. **Specific names** of standard intrinsic functions with corresponding generic names are listed  
 4 in 13.6.

5 If an intrinsic procedure is used as an actual argument to a procedure, its specific name shall be used  
 6 and it may be referenced in the called procedure only with scalar arguments. If an intrinsic procedure  
 7 does not have a specific name, it shall not be used as an actual argument (12.5.1.6).

8 Elemental intrinsic procedures behave as described in 12.8.

## 9 **13.2 Arguments to intrinsic procedures**

### 10 **13.2.1 General rules**

11 All intrinsic procedures may be invoked with either positional arguments or argument keywords (12.5).  
 12 The descriptions in 13.5 through 13.7 give the argument keyword names and positional sequence for  
 13 standard intrinsic procedures.

14 Many of the intrinsic procedures have optional arguments. These arguments are identified by the notation  
 15 “optional” in the argument descriptions. In addition, the names of the optional arguments are enclosed  
 16 in square brackets in description headings and in lists of procedures. The valid forms of reference for  
 17 procedures with optional arguments are described in 12.5.1.

**NOTE 13.3**

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 13.4**

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 (7.4.1.3).

### 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 of collective subroutines**

20 Each argument of 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 invoking `FORM_TEAM` for the team.

## 23 **13.3 Bit model**

24 The bit manipulation procedures and inquiry functions are described in terms of a model for the repre-  
 25 sentation and behaviour of bits on a processor.

26 For an integer, a bit is defined to be a binary digit  $w$  located at position  $k$  of a nonnegative integer scalar  
 27 object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{z-1} w_k \times 2^k$$

28 and for which  $w_k$  may have the value 0 or 1. An example of a model number compatible with the  
 29 examples used in 13.4 would have  $z = 32$ , thereby defining a 32-bit integer.

- 1 Using the notation of the formula above, the value of an object of type bits and kind  $z$  is represented as  
 2 the ordered sequence of bits with  $w_k$  the bit at position  $k$ . The rightmost bit is  $w_0$  and the leftmost bit  
 3 is  $w_{z-1}$ . Such a bits object can be interpreted as a nonnegative integer with the value  $j$ .
- 4 An inquiry function `BIT_SIZE` is available to determine the parameter  $z$  of the model.
- 5 Effectively, this model defines an integer object to consist of  $z$  bits in sequence numbered from right  
 6 to left from 0 to  $z - 1$ . This model is valid only in the context of the use of such an object as the  
 7 argument or result of one of the bit manipulation procedures. In all other contexts, the model defined  
 8 for an integer in 13.4 applies. In particular, whereas the models are identical for  $w_{z-1} = 0$ , they do not  
 9 correspond for  $w_{z-1} = 1$  and the interpretation of bits in such objects is processor dependent.

## 10 13.4 Numeric models

- 11 The numeric manipulation and inquiry functions are described in terms of a model for the representation  
 12 and behavior of numbers on a processor. The model has parameters that are determined so as to make  
 13 the model best fit the machine on which the program is executed.

- 14 The model set for integer  $i$  is defined by

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k$$

- 15 where  $r$  is an integer exceeding one,  $q$  is a positive integer, each  $w_k$  is a nonnegative integer less than  $r$ ,  
 16 and  $s$  is +1 or -1.

- 17 The model set for real  $x$  is defined by

$$x = \begin{cases} 0 \text{ or} \\ s \times b^e \times \sum_{k=1}^p f_k \times b^{-k} , \end{cases}$$

- 18 where  $b$  and  $p$  are integers exceeding one; each  $f_k$  is a nonnegative integer less than  $b$ , with  $f_1$  nonzero;  $s$   
 19 is +1 or -1; and  $e$  is an integer that lies between some integer maximum  $e_{\max}$  and some integer minimum  
 20  $e_{\min}$  inclusively. For  $x = 0$ , its exponent  $e$  and digits  $f_k$  are defined to be zero. The integer parameters  
 21  $r$  and  $q$  determine the set of model integers and the integer parameters  $b$ ,  $p$ ,  $e_{\min}$ , and  $e_{\max}$  determine  
 22 the set of model floating point numbers. The parameters of the integer and real models are available  
 23 for each representation method of the integer and real types. The parameters characterize the set of  
 24 available numbers in the definition of the model. The floating-point manipulation functions (13.5.10)  
 25 and numeric inquiry functions (13.5.6) provide values of some parameters and other values related to  
 26 the models.

### NOTE 13.5

Examples of these functions in 13.7 use the models

$$i = s \times \sum_{k=0}^{30} w_k \times 2^k$$

and



**NOTE 13.5 (cont.)**

$$x = 0 \text{ or } s \times 2^e \times \left( \frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \leq e \leq 127$$

**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 13.6**

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 13.7**

Many of the argument keywords have names that are indicative of their usage. For example:

KIND	Describes the kind type parameter of the result
STRING, STRING_A	An arbitrary character string
BACK	Controls the direction of string scan (forward or backward)
MASK	A mask 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
21	NINT (A [, KIND])	Nearest integer
22	REAL (A [, KIND])	Conversion to real type
23	SIGN (A, B)	Transfer of sign

**24 13.5.2 Mathematical functions**

25	ACOS (X)	Arccosine
26	ACOSH (X)	Hyperbolic arccosine
27	ASIN (X)	Arcsine

1	ASINH (X)	Hyperbolic arcsine
2	ATAN (X) or ATAN (Y, X)	Arctangent
3	ATAN2 (Y, X)	Arctangent
4	ATANH (X)	Hyperbolic arctangent
5	BESSEL_J0 (X)	Bessel function of the first kind of order zero
6	BESSEL_J1 (X)	Bessel function of the first kind of order one
7	BESSEL_JN (N,X)	Bessel function of the first kind of order N
8	BESSEL_Y0 (X)	Bessel function of the second kind of order zero
9	BESSEL_Y1 (X)	Bessel function of the second kind of order one
10	BESSEL_YN (N, X)	Bessel function of the second kind of order N
11	COS (X)	Cosine
12	COSH (X)	Hyperbolic cosine
13	ERF (X)	Error function
14	ERFC (X)	Complementary error function
15	ERFC.SCALED (X)	Exponentially-scaled complementary error function
16		
17	EXP (X)	Exponential
18	GAMMA (X)	Gamma function
19	HYPOT (X, Y)	Euclidean distance function
20	LOG (X)	Natural logarithm
21	LOG_GAMMA (X)	Logarithm of absolute value of gamma function
22	LOG10 (X)	Common logarithm (base 10)
23	SIN (X)	Sine
24	SINH (X)	Hyperbolic sine
25	SQRT (X)	Square root
26	TAN (X)	Tangent
27	TANH (X)	Hyperbolic tangent

### 28 13.5.3 Character functions

29	ACHAR (I [, KIND])	Character in given position in ASCII collating sequence
30		
31	ADJUSTL (STRING)	Adjust left
32	ADJUSTR (STRING)	Adjust right
33	CHAR (I [, KIND])	Character in given position in processor collating sequence
34		
35	IACHAR (C [, KIND])	Position of a character in ASCII collating sequence
36		
37	ICHAR (C [, KIND])	Position of a character in processor collating sequence
38		
39	INDEX (STRING, SUBSTRING [, BACK, KIND])	Starting position of a substring
40	LEN_TRIM (STRING [, KIND])	Length without trailing blank characters
41	LGE (STRING_A, STRING_B)	Lexically greater than or equal
42	LGT (STRING_A, STRING_B)	Lexically greater than
43	LLE (STRING_A, STRING_B)	Lexically less than or equal
44	LLT (STRING_A, STRING_B)	Lexically less than
45	MAX (A1, A2 [, A3,...])	Maximum value
46	MIN (A1, A2 [, A3,...])	Minimum value
47	REPEAT (STRING, NCOPIES)	Repeated concatenation
48	SCAN (STRING, SET [, BACK, KIND])	Scan a string for a character in a set
49	TRIM (STRING)	Remove trailing blank characters
50	VERIFY (STRING, SET [, BACK, KIND])	Verify the set of characters in a string

**1 13.5.4 Kind functions**

2	BITS_KIND (X [, KIND])	Bits kind type parameter value, compatible with
3		the argument
4	KIND (X)	Kind type parameter value
5	SELECTED_BITS_KIND (N)	Bits kind type parameter value, given number of
6		bits
7	SELECTED_CHAR_KIND (NAME)	Character kind type parameter value, given
8		character set name
9	SELECTED_INT_KIND (R)	Integer kind type parameter value, given range
10	SELECTED_REAL_KIND ([P, R, RADIX])	Real kind type parameter value, given precision,
11		range, and radix

**12 13.5.5 Miscellaneous type conversion functions**

13	BITS (A [, KIND])	Convert to bits type
14	LOGICAL (L [, KIND])	Convert between objects of type logical with
15		different kind type parameters
16	TRANSFER (SOURCE, MOLD [, SIZE])	Treat first argument as if of type of second
17		argument

**18 13.5.6 Numeric inquiry functions**

19	DIGITS (X)	Number of significant digits of the model
20	EPSILON (X)	Number that is almost negligible compared to
21		one
22	HUGE (X)	Largest number of the model
23	MAXEXPONENT (X)	Maximum exponent of the model
24	MINEXPONENT (X)	Minimum exponent of the model
25	PRECISION (X)	Decimal precision
26	RADIX (X)	Base of the model
27	RANGE (X)	Decimal exponent range
28	TINY (X)	Smallest positive number of the model

**29 13.5.7 Array inquiry functions**

30	CO_LBOUND (CO_ARRAY [, DIM, KIND])	Lower co-bounds of a co-array
31	CO_UBOUND (CO_ARRAY [, DIM, KIND])	Upper co-bounds of a co-array
32	LBOUND (ARRAY [, DIM, KIND])	Lower dimension bounds of an array
33	SHAPE (SOURCE [, KIND])	Shape of an array or scalar
34	SIZE (ARRAY [, DIM, KIND])	Total number of elements in an array
35	UBOUND (ARRAY [, DIM, KIND])	Upper dimension bounds of an array

**36 13.5.8 Other inquiry functions**

37	ALLOCATED (ARRAY) or ALLOCATED (SCALAR)	Allocation status
38	ASSOCIATED (POINTER [, TARGET])	Association status inquiry or comparison
39	BIT_SIZE (I)	Number of bits of the model
40	EXTENDS_TYPE_OF (A, MOLD)	Same dynamic type or an extension
41	IS_CONTIGUOUS	Contiguity inquiry
42	LEN (STRING [, KIND])	Length of a character entity
43	NEW_LINE (A)	Newline character

1	PRESENT (A)	Argument presence
2	SAME_TYPE_AS (A, B)	Same dynamic type

### 3 13.5.9 Bit manipulation procedures

4	BTEST (I, POS)	Bit testing
5	DSHIFTL (I, J, SHIFT)	Combined left shift
6	DSHIFTR (I, J, SHIFT)	Combined right shift
7	IAND (I, J)	Bitwise AND
8	IBCLR (I, POS)	Clear bit
9	IBITS (I, POS, LEN)	Bit extraction
10	IBSET (I, POS)	Set bit
11	IEOR (I, J)	Exclusive OR
12	IOR (I, J)	Inclusive OR
13	ISHFT (I, SHIFT)	Logical shift
14	ISHFTC (I, SHIFT [, SIZE])	Circular shift
15	LEADZ (I [, KIND])	Number of leading zero bits
16	MASKL (I [, KIND])	Left justified bit mask
17	MASKR (I [, KIND])	Right justified bit mask
18	MERGE_BITS (I, J, MASK)	Merge bits under mask
19	MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	Copies bits from one integer to another
20	NOT (I)	Bitwise complement
21	POPCNT (I [, KIND])	Number of one bits
22	POPPAR (I [, KIND])	Parity of one bits
23	SHIFTA (I, SHIFT)	Arithmetic right shift
24	SHIFTL (I, SHIFT)	Left shift
25	SHIFTR (I, SHIFT)	Right shift
26	TRAILZ (I [, KIND])	Number of trailing zero bits

### 27 13.5.10 Floating-point manipulation functions

28	EXPONENT (X)	Exponent part of a model number
29	FRACTION (X)	Fractional part of a number
30	NEAREST (X, S)	Nearest different processor number in given direction
31		
32	RRSPACING (X)	Reciprocal of the relative spacing of model numbers near given number
33		
34	SCALE (X, I)	Multiply a real by its base to an integer power
35	SET_EXPONENT (X, I)	Set exponent part of a number
36	SPACING (X)	Absolute spacing of model numbers near given number
37		

### 38 13.5.11 Vector and matrix multiply functions

39	DOT_PRODUCT (VECTOR_A, VECTOR_B)	Dot product of two rank-one arrays
40	MATMUL (MATRIX_A, MATRIX_B)	Matrix multiplication

### 41 13.5.12 Array reduction functions

42	ALL (MASK [, DIM])	True if all values are true
43	ANY (MASK [, DIM])	True if any value is true
44	COUNT (MASK [, DIM, KIND])	Number of true elements in an array

1	IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY [, MASK])	Bitwise AND of array elements
2	IANY (ARRAY, DIM [, MASK]) or IANY (ARRAY [, MASK])	Bitwise OR of array elements
3	IPARITY (ARRAY, DIM [, MASK]) or IPARITY (ARRAY [, MASK])	Bitwise exclusive OR of array elements
4	MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])	Maximum value in an array
5	MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])	Minimum value in an array
6	NORM2 (X)	$L_2$ norm of an array
7	PARITY (MASK [, DIM])	True if an odd number of values is true
8	PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])	Product of array elements
9	SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])	Sum of array elements

### 10 13.5.13 Array construction functions

11	CSHIFT (ARRAY, SHIFT [, DIM])	Circular shift
12	EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])	End-off shift
13	MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
14	PACK (ARRAY, MASK [, VECTOR])	Pack an array into an array of rank one under a mask
15	RESHAPE (SOURCE, SHAPE[, PAD, ORDER])	Reshape an array
16	SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
17	TRANSPOSE (MATRIX)	Transpose of an array of rank two
18	UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask
19		
20		

### 21 13.5.14 Array location functions

22	FINDLOC (ARRAY, VALUE, DIM [, MASK, KIND, BACK]) or FINDLOC (ARRAY, VALUE, [, MASK, KIND, BACK])	Location of value in an array
23	MAXLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MAXLOC (ARRAY [, MASK, KIND, BACK])	Location of a maximum value in an array
24	MINLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MINLOC (ARRAY [, MASK, KIND, BACK])	Location of a minimum value in an array

### 25 13.5.15 Collective subroutines

26	CO_ALL (MASK, RESULT [, TEAM])	Whether all values are true
27	CO_ANY (MASK, RESULT [, TEAM])	Whether any value is true
28	CO_COUNT (MASK, RESULT [, TEAM])	Number of true elements in a co-array
29	CO_MAXLOC (CO_ARRAY, RESULT [, TEAM])	Image indices of maximum values
30	CO_MAXVAL (CO_ARRAY, RESULT [, TEAM])	Maximum values

1	CO_MINLOC (CO_ARRAY, RESULT [, TEAM])	Image indices of minimum values
2	CO_MINVAL (CO_ARRAY, RESULT [, TEAM])	Minimum values
3	CO_PRODUCT (CO_ARRAY, RESULT [, TEAM])	Products of elements
4	CO_SUM (CO_ARRAY, RESULT [, TEAM])	Sums of elements
5	FORM_TEAM (TEAM, IMAGES)	Forms a team of images
6	<b>13.5.16 Null function</b>	
7	NULL ([MOLD])	Returns disassociated or unallocated result
8	<b>13.5.17 Allocation transfer procedure</b>	
9	MOVE_ALLOC (FROM, TO)	Moves an allocation from one allocatable object to another
10		
11	<b>13.5.18 Random number subroutines</b>	
12	RANDOM_NUMBER (HARVEST)	Returns pseudorandom number
13	RANDOM_SEED ([SIZE, PUT, GET])	Initializes or restarts the pseudorandom number generator
14		
15	<b>13.5.19 System environment procedures</b>	
16	COMMAND_ARGUMENT_COUNT ()	Number of command arguments
17	CPU_TIME (TIME)	Obtain processor time
18	DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])	Obtain date and time
19	EXECUTE_COMMAND_LINE (COMMAND, [WAIT, EXITSTAT, CMDSTAT, CMDMSG])	Execute a command line
20	GET_COMMAND ([COMMAND, LENGTH, STATUS])	Returns entire command
21	GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])	Returns a command argument
22	GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])	Obtain the value of an environment variable
23	IMAGE_INDEX (CO_ARRAY, SUB)	Index of an image
24	IS_IOSTAT_END (I)	Test for end-of-file value
25	IS_IOSTAT_EOR (I)	Test for end-of-record value
26	NUM_IMAGES ()	Number of images
27	SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])	Obtain data from the system clock
28	THIS_IMAGE ()	Index of the invoking image
29	THIS_IMAGE (CO_ARRAY [, DIM])	Co-subscript list

**J3 internal note****Unresolved Technical Issue 59**

These two uses of THIS\_IMAGE are completely different (to the extent of needing separate summary lines). This is likely to confuse users. Consider changing the name of the second one, perhaps to something like CO\_SUBSCRIPTS, to avoid confusion.

## 1 13.6 Specific names for standard intrinsic functions

- 2 Except for AMAX0, AMIN0, MAX1, and MIN1, the result type of the specific function is the same as the  
 3 result type of the corresponding generic function would be if it were invoked with the same arguments  
 4 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
DPROD	DPROD	default real
DSIGN	SIGN	double precision real
DSIN	SIN	double precision real
DSINH	SINH	double precision real

Specific Name	Generic Name	Argument Type
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  
11 range of values representable by objects of the specified type and type parameters, unless the intrinsic  
12 module IEEE\_ARITHMETIC (clause 14) is accessible and there is support for an infinite or a NaN result,  
13 as appropriate. If an infinite result is returned, the flag IEEE\_OVERFLOW or IEEE\_DIVIDE\_BY\_ZERO  
14 shall signal; if a NaN result is returned, the flag IEEE\_INVALID shall signal. If all results are normal,



1 these flags shall have the same status as when the intrinsic procedure was invoked.

## 2 13.7.1 ABS (A)

3 **Description.** Absolute value.

4 **Class.** Elemental function.

5 **Argument.** A shall be of type integer, real, or complex.

6 **Result Characteristics.** The same as A except that if A is complex, the result is real.

7 **Result Value.** If A is of type integer or real, the value of the result is  $|A|$ ; if A is complex with  
8 value  $(x, y)$ , the result is equal to a processor-dependent approximation to  $\sqrt{x^2 + y^2}$ .

9 **Example.** ABS ((3.0, 4.0)) has the value 5.0 (approximately).

## 10 13.7.2 ACHAR (I [, KIND])

11 **Description.** Returns the character in a specified position of the ASCII collating sequence. It  
12 is the inverse of the IACHAR function.

13 **Class.** Elemental function.

14 **Arguments.**

15 I shall be of type integer or bits. If it is of type bits, it is interpreted as a  
16 nonnegative integer as described in 13.3.

17 KIND (optional) shall be a scalar integer initialization expression.

18 **Result Characteristics.** Character of length one. If KIND is present, the kind type parameter  
19 is that specified by the value of KIND; otherwise, the kind type parameter is that of default  
20 character type.

21 **Result Value.** If I has a value in the range  $0 \leq I \leq 127$ , the result is the character in  
22 position I of the ASCII collating sequence, provided the processor is capable of representing  
23 that character in the character type of the result; otherwise, the result is processor dependent.  
24 ACHAR (IACHAR (C)) shall have the value C for any character C capable of representation in  
the default character type.

25 **Examples.** ACHAR (88) has the value 'X'. ACHAR (Z'41') has the value 'A'.

## 26 13.7.3 ACOS (X)

27 **Description.** Arccosine (inverse cosine) function.

28 **Class.** Elemental function.

29 **Argument.** X shall be of type real with a value that satisfies the inequality  $|X| \leq 1$ , or of type  
30 complex.

31 **Result Characteristics.** Same as X.

32 **Result Value.** The result has a value equal to a processor-dependent approximation to arc-  
33 cos(X). If it is real it is expressed in radians and lies in the range  $0 \leq \text{ACOS}(X) \leq \pi$ . If it is  
34 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     ADJUSTL (STRING)**

3        **Description.** Adjust to the left, removing leading blanks and inserting trailing blanks.

4        **Class.** Elemental function.

5        **Argument.** STRING shall be of type character.

6        **Result Characteristics.** Character of the same length and kind type parameter as STRING.

7        **Result Value.** The value of the result is the same as STRING except that any leading blanks  
8        have been deleted and the same number of trailing blanks have been inserted.

9        **Example.** ADJUSTL (' WORD') has the value 'WORD '.

## 10   **13.7.5     ACOSH (X)**

11       **Description.** Inverse hyperbolic cosine function.

12       **Class.** Elemental function.

13       **Argument.** X shall be of type real or complex.

14       **Result Characteristics.** Same as X.

15       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
16       inverse hyperbolic cosine function of X. If the result is complex the imaginary part is expressed  
17       in radians and lies in the range  $0 \leq \text{AIMAG}(\text{ACOSH}(X)) \leq \pi$

18       **Example.** ACOSH(1.5430806) has the value 1.0 (approximately).

## 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  
25       have 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  $y$ .

**Example.** AIMAG ((2.0, 3.0)) has the value 3.0.

### 13.7.8 AINT (A [, KIND])

**Description.** Truncation to a whole number.

**Class.** Elemental function.

**Arguments.**

A shall be of type real.

KIND (optional) shall be a scalar integer initialization expression.

**Result Characteristics.** The result is of type real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

**Result Value.** If  $|A| < 1$ , AINT (A) has the value 0; if  $|A| \geq 1$ , AINT (A) has a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

**Examples.** AINT (2.783) has the value 2.0. AINT (-2.783) has the value -2.0.

### 13.7.9 ALL (MASK [, DIM])

**Description.** Determine whether all values are true in MASK along dimension DIM.

**Class.** Transformational function.

**Arguments.**

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 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 ALL (MASK) has the value true if all elements of MASK are true or if MASK has size zero, and the result has value false if any element of MASK is false.

*Case (ii):* If MASK has rank one, ALL(MASK,DIM) is equal to ALL(MASK). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of ALL (MASK, DIM) is equal to ALL (MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ ).

**Examples.**

*Case (i):* The value of ALL ((/.TRUE., .FALSE., .TRUE. /)) is false.

*Case (ii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$  and C is the array  $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$  then ALL (B /= C, DIM = 1) is [true, false, false] and ALL (B /= C, DIM = 2) is [false, false].

### 13.7.10 ALLOCATED (ARRAY) or ALLOCATED (SCALAR)

1       **Description.** Indicate whether an allocatable variable is allocated.

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  
8       and 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  
16       is 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,  
18       the result 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  
27       MASK. It is scalar if DIM is absent; otherwise, the result has rank  $n - 1$  and shape  $(d_1, d_2,$   
28        $\dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of MASK.

29       **Result Value.**

30       *Case (i):*       The result of ANY (MASK) has the value true if any element of MASK is true  
31       and has 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,  
33       the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of ANY(MASK, DIM)

1 is equal to ANY(MASK ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ )).

2 **Examples.**

3 *Case (i):* The value of ANY ((/ .TRUE., .FALSE., .TRUE. /)) is true.

4 *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,  
5 DIM = 1) is [true, false, true] and ANY(B /= C, DIM = 2) is [true, true].

6 **13.7.13 ASIN (X)**

7 **Description.** Arcsine (inverse sine) function.

8 **Class.** Elemental function.

9 **Argument.** X shall be of type real with a value that satisfies the inequality  $|X| \leq 1$ , or of type  
10 complex.

11 **Result Characteristics.** Same as X.

12 **Result Value.** The result has a value equal to a processor-dependent approximation to arc-  
13 sin(X). If it is real it is expressed in radians and lies in the range  $-\pi/2 \leq \text{ASIN}(X) \leq$   
14  $\pi/2$ . If it is complex the real part is expressed in radians and lies in the range  $-\pi/2 \leq$   
15  $\text{REAL}(\text{ASIN}(X)) \leq \pi/2$ .

16 **Example.** ASIN (0.84147098) has the value 1.0 (approximately).

17 **13.7.14 ASINH (X)**

18 **Description.** Inverse hyperbolic sine 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  
23 inverse hyperbolic sine function of X. If the result is complex the imaginary part is expressed  
24 in radians and lies in the range  $-\pi/2 \leq \text{AIMAG}(\text{ASINH}(X)) \leq \pi/2$ . If it is real it is expressed  
25 in radians and lies in the range  $-\pi/2 \leq \text{ASIN}(X) \leq \pi/2$ . If it is complex the real part is  
26 expressed in radians and lies in the range  $-\pi/2 \leq \text{REAL}(\text{ASIN}(X)) \leq \pi/2$ .

27 **Example.** ASINH(1.1752012) has the value 1.0 (approximately).

28 **13.7.15 ASSOCIATED (POINTER [, TARGET])**

29 **Description.** Returns the association status of its pointer argument or indicates whether the  
30 pointer is associated with the target.

31 **Class.** Inquiry function.

32 **Arguments.**

33 **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 (7.4.2) in which POINTER is *data-pointer-object* or *proc-pointer-object*. If TARGET is a pointer then its pointer association status shall not be undefined.

**Result Characteristics.** Default logical scalar.

**Result Value.**

- Case (i):* If TARGET is absent, the result is true if POINTER is associated with a target and false if it is not.
- Case (ii):* If TARGET is present and is a procedure, the result is true if POINTER is associated with TARGET.
- Case (iii):* If TARGET is present and is a procedure pointer, the result is true if POINTER and TARGET are associated with the same procedure. If either POINTER or TARGET is disassociated, the result is false.
- Case (iv):* If TARGET is present and is a scalar target, the result is true if TARGET is not a zero-sized storage sequence and the target associated with POINTER occupies the same storage units as TARGET. Otherwise, the result is false. If the POINTER is disassociated, the result is false.
- Case (v):* If TARGET is present and is an array target, the result is true if the target associated with POINTER and TARGET have the same shape, are neither of size zero nor arrays whose elements are zero-sized storage sequences, and occupy the same storage units in array element order. Otherwise, the result is false. If POINTER is disassociated, the result is false.
- Case (vi):* If TARGET is present and is a scalar pointer, the result is true if the target associated with POINTER and the target associated with TARGET are not zero-sized storage sequences and they occupy the same storage units. Otherwise, the result is false. If either POINTER or TARGET is disassociated, the result is false.
- Case (vii):* If TARGET is present and is an array pointer, the result is true if the target associated with POINTER and the target associated with TARGET have the same shape, are neither of size zero nor arrays whose elements are zero-sized storage sequences, and occupy the same storage units in array element order. Otherwise, the result is false. If either POINTER or TARGET is disassociated, the result is false.

**Examples.** ASSOCIATED (CURRENT, HEAD) is true if CURRENT is associated with the target HEAD. After the execution of

```
A_PART => A (:N)
```

ASSOCIATED (A\_PART, A) is true if N is equal to UBOUND (A, DIM = 1). After the execution of

```
NULLIFY (CUR); NULLIFY (TOP)
```

ASSOCIATED (CUR, TOP) is false.

### 13.7.16 ATAN (X) or ATAN (Y, X)

**Description.** Arctangent (inverse tangent) function.

**Class.** Elemental function.

**Arguments.**

1           Y                   shall be of type real.

          X                   If Y is present, X shall be of type real with the same kind type parameter as  
2                               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.

3       **Result Characteristics.** Same as X.

4       **Result Value.** If Y is present, the result is the same as the result of ATAN2(Y,X). If Y is  
5       absent, the result has a value equal to a processor-dependent approximation to  $\arctan(X)$  whose  
6       real part is expressed in radians and lies in the range  $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$ .

7       **Example.** ATAN (1.5574077) has the value 1.0 (approximately).

### 8   13.7.17    **ATAN2 (Y, X)**

9       **Description.** Arctangent (inverse tangent) function. The result is the principal value of the  
10       argument of the nonzero complex number (X, Y).

11       **Class.** Elemental function.

12       **Arguments.**

13       Y                   shall be of type real.

14       X                   shall be of the same type and kind type parameter as Y. If Y has the value  
                              zero, X shall not have the value zero.

15       **Result Characteristics.** Same as X.

16       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
17       principal value of the argument of the complex number (X, Y), expressed in radians. It lies in  
18       the range  $-\pi \leq \text{ATAN2}(Y,X) \leq \pi$  and is equal to a processor-dependent approximation to a  
19       value of  $\arctan(Y/X)$  if  $X \neq 0$ . If  $Y > 0$ , the result is positive. If  $Y = 0$  and  $X > 0$ , the result  
20       is Y. If  $Y = 0$  and  $X < 0$ , then the result is  $\pi$  if Y is positive real zero or the processor cannot  
21       distinguish between positive and negative real zero, and  $-\pi$  if Y is negative real zero. If  $Y < 0$ ,  
22       the result is negative. If  $X = 0$ , the absolute value of the result is  $\pi/2$ .

23       **Examples.** ATAN2 (1.5574077, 1.0) has the value 1.0 (approximately). If Y has the value  
24        $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$  and X has the value  $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$ , the value of ATAN2 (Y, X) is approximately  
25        $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ -\frac{3\pi}{4} & -\frac{\pi}{4} \end{bmatrix}$ .

### 26   13.7.18    **ATANH (X)**

27       **Description.** Inverse hyperbolic tangent function.

28       **Class.** Elemental function.

29       **Argument.** X shall be of type real or complex.

30       **Result Characteristics.** Same as X.

31       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
32       inverse hyperbolic tangent function of X. If the result is complex the imaginary part is expressed  
33       in radians and lies in the range  $-\pi/2 \leq \text{AIMAG}(\text{ATANH}(X)) \leq \pi/2$ .

1       **Example.** ATANH(0.76159416) has the value 1.0 (approximately).

## 2   **13.7.19    BESSEL\_J0 (X)**

3       **Description.** Bessel function of the first kind of order zero.

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  
8       Bessel function of the first kind of order zero of X.

9       **Example.** BESSEL\_J0 (1.0) has the value 0.765 (approximately).

## 10   **13.7.20    BESSEL\_J1 (X)**

11       **Description.** Bessel function of the first kind of order one.

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  
16       Bessel function of the first kind of order one of X.

17       **Example.** BESSEL\_J1 (1.0) has the value 0.440 (approximately).

## 18   **13.7.21    BESSEL\_JN (N, X)**

19       **Description.** Bessel function of the first kind of order N.

20       **Class.** Elemental function.

21       **Arguments.**

22       N               shall be of type integer and nonnegative.

23       X               shall be of type real.

24       **Result Characteristics.** Same as X.

25       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
26       Bessel function of the first kind of order N of X.

27       **Example.** BESSEL\_JN (2, 1.0) has the value 0.115 (approximately).

## 28   **13.7.22    BESSEL\_Y0 (X)**

29       **Description.** Bessel function of the second kind of order zero.

30       **Class.** Elemental function.

31       **Argument.** X shall be of type real. Its value shall be greater than zero.

32       **Result Characteristics.** Same as X.



1       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
 2       Bessel function of the second kind of order zero of X.

3       **Example.** BESSEL\_Y0(1.0) has the value 0.088 (approximately).

#### 4   **13.7.23    BESSEL\_Y1 (X)**

5       **Description.** Bessel function of the second kind of order one.

6       **Class.** Elemental function.

7       **Argument.** 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  
 10       Bessel function of the second kind of order one of X.

11       **Example.** BESSEL\_Y1 (1.0) has the value -0.781 (approximately).

#### 12   **13.7.24    BESSEL\_YN (N, X)**

13       **Description.** Bessel function of the second kind of order N.

14       **Class.** Elemental function.

15       **Arguments.**

16       N               shall be of type integer and nonnegative.

17       X               shall be of type real. Its value shall be greater than zero.

18       **Result Characteristics.** Same as X.

19       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
 20       Bessel function of the second kind of order N of X.

21       **Example.** BESSEL\_YN (2, 1.0) has the value -1.651 (approximately).

#### 22   **13.7.25    BIT\_SIZE (I [, KIND])**

##### **J3 internal note**

##### **Unresolved Technical Issue 62**

Another totally useless ineffective KIND argument. While intrinsic KINDs are default integer, this is a waste of effort.

<RANT>

And what half-baked idea is this, that we provide for bits capabilities for strings not only up to, but *beyond* 512MB(!) but oh no, it's much too hard to have *variable-length* ones?!?! What on earth are you thinking; only a complete loon would implement more than 4 billion different lengths of bitstring without building variable-length infrastructure.

</RANT>

23       **Description.** Returns the number of bits  $z$  defined by the model of 13.3.

24       **Class.** Inquiry function.

25       **Arguments.**

1 I shall be of type integer or bits. It may be a scalar or an array.

2 KIND (optional) shall be a scalar integer initialization expression.

3 **Result Characteristics.** Scalar integer. If KIND is present, the kind type parameter is that  
4 specified by the value of KIND. If KIND is not present and I is of type integer the result is a  
5 scalar integer with the same kind type parameter as I. If KIND is not present and I is of type  
6 bits, the result is a scalar of type default integer.

7 **Result Value.** If I is of type integer, result has the value of the number of bits  $z$  of the model  
8 integer defined for bit manipulation contexts in 13.3. If I is of type bits, the result has the value  
9 KIND(I).

10 **Examples.** BIT\_SIZE (1) has the value 32 if  $z$  of the model is 32. BIT\_SIZE(Z'00FF') has the  
11 value 16.

## 12 13.7.26 BITS (A [, KIND])

13 **Description.** Convert to bits type.

14 **Class.** Elemental function.

15 **Arguments.**

16 A shall be of type integer, real, complex, logical, or bits.

17 KIND (optional) shall be a scalar integer initialization expression.

18 **Result Characteristics.** Bits. If KIND is present, the kind type parameter is that specified  
19 by the value of KIND; otherwise, the kind type parameter is BITS\_KIND(A).

20 **Result Value.** If A is of type bits and the kind type parameter of A is the same as that of the  
21 result, the result value is the value of A.

22 If A is of type bits with a smaller kind type parameter value than that of the result, the rightmost  
23 KIND(A) bits of the result value are the same as those of A and the remaining bits of the result are 0.

24 If A is of type bits with a larger kind type parameter value than that of the result, the result value  
25 consists of the rightmost KIND(*result*) bits of A.

26 If A is not of type bits and BITS\_KIND(A) is greater than or equal to the kind type parameter of the  
27 result, the result value consists of the rightmost KIND(*result*) bits of the physical representation of A.

28 If A is not of type bits and BITS\_KIND(A) is less than or equal to the kind type parameter of the result,  
29 the rightmost bits of the result are those of the physical representation of A and the remaining bits of  
30 the result are 0.

31 **Examples.** BITS (Z'ABCD', 32) has the value Z'0000ABCD'. BITS (−1) has the value  
32 Z'FFFFFFFF' for a processor whose default integer representation is 32-bit two's-complement.  
33 BITS (.TRUE., 5) has the value B'00001' for a processor that represents the logical value true  
34 by setting the low order bit of the internal value and clearing the other bits. BITS (X) has the  
35 value Z'7F800000' if the value of X is an IEEE single precision positive infinity.

## 36 13.7.27 BITS\_KIND (X [, KIND])

37 **Description.** Return bits kind compatible with the argument.

38 **Class.** Inquiry function.

**Arguments.**

X shall be of type bits, integer, real, complex, or logical.

KIND (optional) shall be a scalar integer initialization expression.

**Result Characteristics.** Scalar 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.

**J3 internal note**

## Unresolved Technical Issue 60

Maybe it is good that people are thinking about bitstrings greater than 512MB, but since intrinsic kind type parameters are of default integer type, there is a problem here. Two choices:

- (1) make the kind type parameter for bits type processor-dependent or something, *AND* change the KIND intrinsic to have a KIND argument (bleurgh), or
- (2) remove the useless KIND argument from this intrinsic since it will indeed be useless.

**Result Characteristics.** If X is of type bits, the result has the value KIND (X). If X is of type default integer, default real, or default logical, the result has the value NUMERIC\_STORAGE\_SIZE (13.8.3.13). If X is of type double precision or default complex, the result has the value  $2 \times \text{NUMERIC\_STORAGE\_SIZE}$ . If X is of type complex with the same kind type parameter as that of double precision, the result has the value  $4 \times \text{NUMERIC\_STORAGE\_SIZE}$ . If X is of type non-default integer, the result has the value BIT\_SIZE (X). If X is of a non-default logical or non-default non-integer numeric type, the result value is the number of bits of storage used by the processor to represent scalar objects of that type and kind.

**J3 internal note**

## Unresolved Technical Issue 61

COMPLEX(KIND(0d0)) is neither required nor guaranteed to take double the space of DOUBLE PRECISION; it takes a single (unique) storage unit.

BIT\_SIZE(X) does not return the number of bits taken for the storage of X, and indeed the earlier Cray models had “small integers” which took the full 60 bits of storage but were only calculated to rather fewer (something like 32ish and 48ish IIRC).

Machines which use a unified FP/Integer representation will also have a BIT\_SIZE smaller than the actual storage.

Indeed, BIT\_SIZE(default integer) is neither required nor guaranteed to be the same as NUMERIC\_STORAGE\_SIZE.

Therefore the sentence about non-default integers should be struck, as well as the sentence about double precision complex.

**Example.** The value of BITS\_KIND (0) is 32 if the size of a numeric storage unit is 32 bits.

**13.7.28 BTEST (I, POS)**

**Description.** Tests a bit of an integer or bits value.

**Class.** Elemental function.

**Arguments.**

I shall be of type integer or bits.

POS shall be of type integer. It shall be nonnegative and be less than BIT\_SIZE (I).

**Result Characteristics.** Default logical.

**Result Value.** The result has the value true if bit POS of I has the value 1 and has the value false if bit POS of I has the value 0. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

**Examples.** BTEST (8, 3) has the value true. If A has the value  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ , the value of BTEST (A, 2) is  $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$  and the value of BTEST (2, A) is  $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$ . BTEST (B'01000', 3) has the value true.

### 13.7.29 CEILING (A [, KIND])

**Description.** Returns the least integer greater than or equal to its argument.

**Class.** Elemental function.

**Arguments.**

A shall be of type real.

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.

**Result Value.** The result has a value equal to the least integer greater than or equal to A.

**Examples.** CEILING (3.7) has the value 4. CEILING (−3.7) has the value −3.

### 13.7.30 CHAR (I [, KIND])

**Description.** Returns the character in a given position of the processor collating sequence associated with the specified kind type parameter. It is the inverse of the ICHAR function.

**Class.** Elemental function.

**Arguments.**

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.

KIND (optional) shall be a scalar integer initialization expression.

**Result Characteristics.** Character of length one. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default character type.

**Result Value.** The result is the character in position I of the collating sequence associated with the specified kind type parameter. ICHAR (CHAR (I, KIND (C))) shall have the value I for  $0 \leq I \leq n - 1$  and CHAR (ICHAR (C), KIND (C)) shall have the value C for any character C capable of representation in the processor.

**Examples.** CHAR (88) has the value 'X' on a processor using the ASCII collating sequence for the default character type. CHAR (Z'41') has the value 'A' on a processor using the ASCII

1 collating sequence.

### 2 **13.7.31 CMPLX (X [, Y, KIND])**

3 **Description.** Convert to complex type.

4 **Class.** Elemental function.

5 **Arguments.**

6 X shall be of type integer, real, or complex, or bits.

7 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.

8 KIND (optional) shall be a scalar integer initialization expression.

9 **Result Characteristics.** The result is of type complex. If KIND is present, the kind type  
10 parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of  
11 default real type.

12 **Result Value.** If Y is absent and X is not complex, it is as if Y were present with the value  
13 zero. If X is complex, it is as if X were real with the value REAL (X, KIND) and Y were present  
14 with the value AIMAG (X, KIND). CMPLX (X, Y, KIND) has the complex value whose real  
15 part is REAL (X, KIND) and whose imaginary part is REAL (Y, KIND).

16 **Example.** CMPLX (-3) has the value (-3.0, 0.0).

### 17 **13.7.32 COMMAND\_ARGUMENT\_COUNT ()**

18 **Description.** Returns the number of command arguments.

19 **Class.** Inquiry function.

20 **Argument.** None.

21 **Result Characteristics.** Scalar default integer.

22 **Result Value.** The result value is equal to the number of command arguments available.  
23 If there are no command arguments available or if the processor does not support command  
24 arguments, then the result has the value zero. If the processor has a concept of a command  
25 name, the command name does not count as one of the command arguments.

26 **Example.** See 13.7.73.

### 27 **13.7.33 CO\_ALL (MASK, RESULT [, TEAM])**

28 **Description.** Determine whether all values are true on a team of images.

29 **Class.** Collective subroutine.

30 **Arguments.**

31 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 (13.8.3.5). 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 the statement CALL CO\_ALL (MASK, RESULT) is [true, false, true].

### 13.7.34 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 (13.8.3.5). 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 the statement CALL CO\_ANY (MASK, RESULT) is [true, true, false].

### 13.7.35 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 (13.8.3.5). 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 the statement CALL CO\_COUNT (MASK, RESULT) is [2, 1, 0].

### 13.7.36 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.

*Case (ii):* CO\_LBOUND (CO\_ARRAY) has a value whose  $i^{th}$  element is equal to CO\_LBOUND (CO\_ARRAY,  $i$ ), for  $i = 1, 2, \dots, n$ , where  $n$  is the co-rank of CO\_ARRAY.

**Examples.** If A is allocated by the statement ALLOCATE (A [2:3, 7:\*]) then CO\_LBOUND (A) is [2, 7] and CO\_LBOUND (A, DIM=2) is 7.

### 13.7.37 CO\_MAXLOC (CO\_ARRAY, RESULT [, TEAM])

**Description.** Image indices of the maximum 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 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. 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 (13.8.3.5). It is an INTENT(IN) argument that specifies the team for which CO\_MAXLOC 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 the statement CALL CO\_MAXLOC (CO\_ARRAY, RESULT) is [2, 1, 1].

### 13.7.38 CO\_MAXVAL (CO\_ARRAY, RESULT [, TEAM])

**Description.** Maximum 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 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.

TEAM (optional) shall be a scalar of type IMAGE\_TEAM (13.8.3.5). 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 the statement CALL CO\_MAXVAL (CO\_ARRAY, RESULT) is [4, 5, 6].

### 13.7.39 CO\_MINLOC (CO\_ARRAY, RESULT [, TEAM])

**Description.** Image indices of the minimum values of the elements on a team of images.

**Class.** Collective subroutine.



1      **Arguments.**

2      **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.

3      **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.

4      **TEAM** (optional) shall be a scalar of type IMAGE\_TEAM (13.8.3.5). 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.

5      **Example.** If the number of images is two and CO\_ARRAY is the array [1, 5, 6] on one image  
6      and [4, 1, 6] on the other image, the value of RESULT after the statement  
7      CALL CO\_MINLOC (ARRAY, RESULT) is [1, 2, 1].

8      **13.7.40      CO\_MINVAL (CO\_ARRAY, RESULT [, TEAM])**

9      **Description.** Minimum values of the elements on a team of images.

10      **Class.** Collective subroutine.

11      **Arguments.**

12      **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.

13      **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.

14      **TEAM** (optional) shall be a scalar of type IMAGE\_TEAM (13.8.3.5). 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.

15      **Example.** If the number of images is two and CO\_ARRAY is the array [1, 5, 3] on one image  
16      and [4, 1, 6] on the other image, the value of RESULT after the statement  
17      CALL CO\_MINVAL (CO\_ARRAY, RESULT) is [1, 1, 3].

18      **13.7.41      CO\_PRODUCT (CO\_ARRAY, RESULT [, TEAM])**

1       **Description.** Products of elements on a team of images.

2       **Class.** Collective subroutine.

3       **Arguments.**

4       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.

5       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.

6       TEAM (optional) shall be a scalar of type IMAGE\_TEAM (13.8.3.5). 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.

7       **Example.** If the number of images is two and CO\_ARRAY is the array [1, 5, 3] on one image  
8       and [4, 1, 6] on the other image, the value of RESULT after the statement  
9       CALL CO\_PRODUCT (CO\_ARRAY, RESULT) is [4, 5, 18].

## 10   **13.7.42   CO\_SUM (CO\_ARRAY, RESULT [, TEAM])**

11       **Description.** Sums of elements on a team of images.

12       **Class.** Collective subroutine.

13       **Arguments.**

14       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.

15       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.

16       TEAM (optional) shall be a scalar of type IMAGE\_TEAM (13.8.3.5). 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.

17       **Example.** If the number of images is two and CO\_ARRAY is the array [1, 5, 3] on one image  
18       and [4, 1, 6] on the other image, the value of RESULT after the statement  
19       CALL CO\_SUM (CO\_ARRAY, RESULT) is [5, 6, 9].

## 20   **13.7.43   CO\_UBOUND (CO\_ARRAY [, DIM, KIND])**

21       **Description.** Returns all the upper co-bounds or a specified upper co-bound of a co-array of  
22       co-rank greater than one.

1        **Class.** Inquiry function.

2        **Arguments.**

3        CO\_ARRAY        shall be a co-array of any type and of co-rank greater than one. It may be a scalar or an array. If it is allocatable it shall be allocated.

4        DIM (optional)    d 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.

5        KIND (optional)   shall be a scalar integer initialization expression.

6        **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  $n - 1$ , where  $n$  is the co-rank of CO\_ARRAY.

10       **Result Value.**

11       *Case (i):*        CO\_UBOUND (CO\_ARRAY, DIM) has a value equal to the upper co-bound for co-subscript DIM of CO\_ARRAY.

12       *Case (ii):*       CO\_UBOUND (CO\_ARRAY) has a value whose  $i^{th}$  element is equal to CO\_UBOUND (CO\_ARRAY,  $i$ ), for  $i = 1, 2, \dots, n - 1$ , where  $n$  is the co-rank of CO\_ARRAY.

16       **Examples.** If A is allocated by the statement ALLOCATE (A [2:3, 0:7, \*]) then CO\_UBOUND (A) is [3, 7] and CO\_UBOUND (A, DIM=2) is 7.

## 18    13.7.44    CONJG (Z)

19       **Description.** Conjugate of a complex number.

20       **Class.** Elemental function.

21       **Argument.** Z shall be of type complex.

22       **Result Characteristics.** Same as Z.

23       **Result Value.** If Z has the value  $(x, y)$ , the result has the value  $(x, -y)$ .

24       **Example.** CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

## 25    13.7.45    COS (X)

26       **Description.** Cosine function.

27       **Class.** Elemental function.

28       **Argument.** X shall be of type real or complex.

29       **Result Characteristics.** Same as X.

30       **Result Value.** The result has a value equal to a processor-dependent approximation to  $\cos(X)$ . If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

1       **Example.** COS (1.0) has the value 0.54030231 (approximately).

## 2   **13.7.46    COSH (X)**

3       **Description.** 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 cosh(X).  
8       If X is of type complex its imaginary part is regarded as a value in radians.

9       **Example.** COSH (1.0) has the value 1.5430806 (approximately).

## 10   **13.7.47    COUNT (MASK [, DIM, KIND])**

11       **Description.** Count the number of true elements of MASK along dimension DIM.

12       **Class.** Transformational function.

13       **Arguments.**

14       MASK               shall be of type logical. It shall be an array.

15       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.

16       KIND (optional)   shall be a scalar integer initialization expression.

17       **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.  
18       The result 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.  
19       20

21       **Result Value.**

22       *Case (i):*       The result of COUNT (MASK) has a value equal to the number of true elements  
23       of MASK or has the value zero if MASK has size zero.

24       *Case (ii):*     If MASK has rank one, COUNT (MASK, DIM) has a value equal to that of  
25       COUNT (MASK). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1},$   
26        $\dots, s_n)$  of COUNT (MASK, DIM) is equal to COUNT (MASK  $(s_1, s_2, \dots,$   
27        $s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ ).

28       **Examples.**

29       *Case (i):*       The value of COUNT ((/ .TRUE., .FALSE., .TRUE. /)) is 2.

30       *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,  
31       DIM = 1) is [2, 0, 1] and COUNT (B /= C, DIM = 2) is [1, 2].

## 32   **13.7.48    CPU\_TIME (TIME)**

33       **Description.** Returns the processor time.

34       **Class.** Subroutine.

**Argument.** TIME shall be scalar and of type real. It is an INTENT(OUT) argument that is assigned a processor-dependent approximation to the processor time in seconds. If the processor cannot return a meaningful time, a processor-dependent negative value is returned.

**Example.**

```

REAL T1, T2
...
CALL CPU_TIME(T1)
... ! Code to be timed.
CALL CPU_TIME(T2)
WRITE (*,*) 'Time taken by code was ', T2-T1, ' seconds'
```

writes the processor time taken by a piece of code.

#### NOTE 13.8

A processor for which a single result is inadequate (for example, a parallel processor) might choose to provide an additional version for which time is an array.

The exact definition of time is left imprecise because of the variability in what different processors are able to provide. The primary purpose is to compare different algorithms on the same processor or discover which parts of a calculation are the most expensive.

The start time is left imprecise because the purpose is to time sections of code, as in the example.

Most computer systems have multiple concepts of time. One common concept is that of time expended by the processor for a given program. This might or might not include system overhead, and has no obvious connection to elapsed “wall clock” time.

### 13.7.49 CSHIFT (ARRAY, SHIFT [, DIM])

**Description.** Perform a circular shift on an array expression of rank one or perform circular shifts on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions.

**Class.** Transformational function.

**Arguments.**

ARRAY            may be of any type. It shall be an array.

SHIFT            shall be of type integer and shall be scalar if ARRAY has rank one; otherwise, it shall be scalar or of rank  $n - 1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

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 is of the type and type parameters of ARRAY, and has the shape of ARRAY.

**Result Value.**

- 1      *Case (i):*      If ARRAY has rank one, element  $i$  of the result is ARRAY (1 + MODULO ( $i$  +  
2                      SHIFT - 1, SIZE (ARRAY))).
- 3      *Case (ii):*      If ARRAY has rank greater than one, section ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1},$   
4                       $\dots, s_n$ ) of the result has a value equal to CSHIFT (ARRAY ( $s_1, s_2, \dots, s_{\text{DIM}-1},$   
5                       $:, s_{\text{DIM}+1}, \dots, s_n$ ),  $sh, 1$ ), where  $sh$  is SHIFT or SHIFT ( $s_1, s_2, \dots, s_{\text{DIM}-1},$   
6                       $s_{\text{DIM}+1}, \dots, s_n$ ).

### 7      **Examples.**

8      *Case (i):*      If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V circularly to the left by  
9                      two positions is achieved by CSHIFT (V, SHIFT = 2) which has the value [3, 4,  
10                     5, 6, 1, 2]; CSHIFT (V, SHIFT = -2) achieves a circular shift to the right by two  
11                     positions and has the value [5, 6, 1, 2, 3, 4].

12      *Case (ii):*      The rows of an array of rank two may all be shifted by the same amount or by

different amounts. If M is the array  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ , the value of

CSHIFT (M, SHIFT = -1, DIM = 2) is  $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$ , and the value of

CSHIFT (M, SHIFT = (/ -1, 1, 0 /), DIM = 2) is  $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$ .

## 16      **13.7.50      DATE\_AND\_TIME ([DATE, TIME, ZONE, VALUES])**

17      **Description.** Returns data about the real-time clock and date in a form compatible with the  
18                      representations defined in ISO 8601:1988.

19      **Class.** Subroutine.

### 20      **Arguments.**

DATE (optional) shall be scalar and of type default character. It is an INTENT (OUT) argu-  
ment. It is assigned a value of the form *CCYYMMDD*, where *CC* is the  
century, *YY* is the year within the century, *MM* is the month within the year,  
and *DD* is the day within the month. If there is no date available, DATE is  
assigned all blanks.

TIME (optional) shall be scalar and of type default character. It is an INTENT (OUT) argu-  
ment. It is assigned a value of the form *hhmmss.sss*, where *hh* is the hour  
of the day, *mm* is the minutes of the hour, and *ss.sss* is the seconds and  
milliseconds of the minute. If there is no clock available, TIME is assigned  
all blanks.

ZONE (optional) shall be scalar and of type default character. It is an INTENT (OUT) argu-  
ment. It is assigned a value of the form *+hhmm* or *-hhmm*, where *hh* and *mm*  
are the time difference with respect to Coordinated Universal Time (UTC) in  
hours and minutes, respectively. If this information is not available, ZONE  
is assigned all blanks.

VALUES  
(optional) shall be of type default integer and of rank one. It is an INTENT (OUT)  
argument. 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 available;

- 1           VALUES (2)    the month of the year, or –HUGE (0) if there is no date available;  
 2           VALUES (3)    the day of the month, or –HUGE (0) if there is no date available;  
 3           VALUES (4)    the time difference with respect to Coordinated Universal Time (UTC) in  
                           minutes, or –HUGE (0) if this information is not available;  
 4           VALUES (5)    the hour of the day, in the range of 0 to 23, or –HUGE (0) if there is no clock;  
 5           VALUES (6)    the minutes of the hour, in the range 0 to 59, or –HUGE (0) if there is no  
                           clock;  
 6           VALUES (7)    the seconds of the minute, in the range 0 to 60, or –HUGE (0) if there is no  
                           clock;  
 7           VALUES (8)    the milliseconds of the second, in the range 0 to 999, or –HUGE (0) if there  
                           is no clock.

8           **Example.**

```

9           INTEGER DATE_TIME (8)
10          CHARACTER (LEN = 10) BIG_BEN (3)
11          CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2), &
12                               BIG_BEN (3), DATE_TIME)
```

13           If run in Geneva, Switzerland on April 12, 1985 at 15:27:35.5 with a system configured for the  
 14           local time zone, this sample would have assigned the value 19850412 to BIG\_BEN (1), the value  
 15           152735.500 to BIG\_BEN (2), the value +0100 to BIG\_BEN (3), and the value (/ 1985, 4, 12,  
 16           60, 15, 27, 35, 500 /) to DATE\_TIME.

**NOTE 13.9**

UTC is defined by ISO 8601:1988.

17 **13.7.51    DBLE (A)**

18           **Description.** Convert to double precision real type.

19           **Class.** Elemental function.

20           **Argument.** A shall be of type integer, real, or complex, or bits.

21           **Result Characteristics.** Double precision real.

22           **Result Value.** The result has the value REAL (A, KIND (0.0D0)).

23           **Example.** DBLE (–3) has the value –3.0D0.

24 **13.7.52    DIGITS (X)**

25           **Description.** Returns the number of significant digits of the model representing numbers of  
 26           the same type and kind type parameter as the argument.

27           **Class.** Inquiry function.

28           **Argument.** X shall be of type integer or real. It may be a scalar or an array.

29           **Result Characteristics.** Default integer scalar.

1       **Result Value.** The result has the value  $q$  if  $X$  is of type integer and  $p$  if  $X$  is of type real,  
 2       where  $q$  and  $p$  are as defined in 13.4 for the model representing numbers of the same type and  
 3       kind type parameter as  $X$ .

4       **Example.** DIGITS ( $X$ ) has the value 24 for real  $X$  whose model is as in Note 13.5.

### 5   13.7.53    DIM ( $X$ , $Y$ )

6       **Description.** The difference  $X-Y$  if it is positive; otherwise zero.

7       **Class.** Elemental function.

8       **Arguments.**

9        $X$                    shall be of type integer or real.

10       $Y$                    shall be of the same type and kind type parameter as  $X$ .

11      **Result Characteristics.** Same as  $X$ .

12      **Result Value.** The value of the result is  $X-Y$  if  $X>Y$  and zero otherwise.

13      **Example.** DIM ( $-3.0$ ,  $2.0$ ) has the value  $0.0$ .

### 14   13.7.54    DOT\_PRODUCT (VECTOR\_A, VECTOR\_B)

15      **Description.** Performs dot-product multiplication of numeric or logical vectors.

16      **Class.** Transformational function.

17      **Arguments.**

18      VECTOR\_A           shall be of numeric type (integer, real, or complex) or of logical type. It shall  
                              be a rank-one array.

19      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.

20      **Result Characteristics.** If the arguments are of numeric type, the type and kind type pa-  
 21      rameter of the result are those of the expression VECTOR\_A \* VECTOR\_B determined by the  
 22      types of the arguments according to 7.1.4.2. If the arguments are of type logical, the result is  
 23      of type logical with the kind type parameter of the expression VECTOR\_A .AND. VECTOR\_B  
 24      according to 7.1.4.2. The result is scalar.

25      **Result Value.**

26      *Case (i):*       If VECTOR\_A is of type integer or real, the result has the value SUM (VEC-  
 27                       TOR\_A\*VECTOR\_B). If the vectors have size zero, the result has the value zero.

28      *Case (ii):*     If VECTOR\_A is of type complex, the result has the value SUM (CONJG (VEC-  
 29                       TOR\_A)\*VECTOR\_B). If the vectors have size zero, the result has the value  
 30                       zero.

31      *Case (iii):*    If VECTOR\_A is of type logical, the result has the value ANY (VECTOR\_A  
 32                       .AND. VECTOR\_B). If the vectors have size zero, the result has the value false.

33      **Example.** DOT\_PRODUCT ((/ 1, 2, 3 /), (/ 2, 3, 4 /)) has the value 20.

### 34   13.7.55    DPROD ( $X$ , $Y$ )



1       **Description.** Double precision real product.

2       **Class.** Elemental function.

3       **Arguments.**

4       X                   shall be of type default real.

5       Y                   shall be of type default real.

6       **Result Characteristics.** Double precision real.

7       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
8       product of X and Y.

9       **Example.** DPROD (−3.0, 2.0) has the value −6.0D0.

## 10   **13.7.56   DSHIFTL (I, J, SHIFT)**

11       **Description.** Performs a combined left shift.

12       **Class.** Elemental function.

13       **Arguments.**

14       I                   shall be of type integer or bits.

15       J                   shall be of the same type and kind as I.

16       SHIFT               shall be of type integer. It shall be nonnegative and less than or equal to  
                              BIT\_SIZE (I).

17       **Result Characteristics.** Same as I.

18       **Result Value.** The result has the value  
19       IOR (SHIFTL (I, SHIFT), SHIFTR (J, BIT\_SIZE (J)−SHIFT)). DSHIFTL (I, I, SHIFT) has  
20       the same result value as ISHFTC (I, SHIFT).

21       **Example.** DSHIFTL (Z'01234567', Z'89ABCDEF', 8) has the value Z'23456789'.

## 22   **13.7.57   DSHIFTR (I, J, SHIFT)**

23       **Description.** Performs a combined right shift.

24       **Class.** Elemental function.

25       **Arguments.**

26       I                   shall be of type integer or bits.

27       J                   shall be of the same type and kind as I.

28       SHIFT               shall be of type integer. It shall be nonnegative and less than or equal to  
                              BIT\_SIZE (I).

29       **Result Characteristics.** Same as I.

30       **Result Value.** The result has the value IOR (SHIFTL (I, BIT\_SIZE (I)−SHIFT), SHIFTR (J, SHIFT)).  
31       DSHIFTR (I, I, SHIFT) has the same result value as ISHFTC (I, −SHIFT).

32       **Examples.** DSHIFTR(Z'01234567', Z'89ABCDEF', 8) has the value Z'6789ABCD'.

1 DSHIFTR (B'111', B'000', 2) has the value B'110'.

## 2 13.7.58 EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])

3 **Description.** Perform an end-off shift on an array expression of rank one or perform end-off  
4 shifts on all the complete rank-one sections along a given dimension of an array expression of  
5 rank two or greater. Elements are shifted off at one end of a section and copies of a boundary  
6 value are shifted in at the other end. Different sections may have different boundary values and  
7 may be shifted by different amounts and in different directions.

8 **Class.** Transformational function.

9 **Arguments.**

10 **ARRAY** may be of any type. It shall be an array.

11 **SHIFT** shall be of type integer and shall be scalar if ARRAY has rank one; otherwise,  
it shall be scalar or of rank  $n - 1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

**BOUNDARY** shall be of the same type and type parameters as ARRAY and shall be scalar  
(optional) 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.

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

### 12 J3 internal note

Unresolved Technical Issue 64

No default padding value in EOSHIFT for type Bits. Is this deliberate?

13 **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.

14 **Result Characteristics.** The result has the type, type parameters, and shape of ARRAY.

15 **Result Value.** Element  $(s_1, s_2, \dots, s_n)$  of the result has the value  $\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1},$   
16  $s_{\text{DIM}} + sh, s_{\text{DIM}+1}, \dots, s_n)$  where  $sh$  is SHIFT or SHIFT  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$   
17 provided the inequality  $\text{LBOUND}(\text{ARRAY}, \text{DIM}) \leq s_{\text{DIM}} + sh \leq \text{UBOUND}(\text{ARRAY}, \text{DIM})$   
18 holds and is otherwise BOUNDARY or BOUNDARY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ .

19 **Examples.**

20 *Case (i):* If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V end-off to the left by 3  
21 positions is achieved by EOSHIFT (V, SHIFT = 3), which has the value [4, 5,  
22 6, 0, 0, 0]; EOSHIFT (V, SHIFT = -2, BOUNDARY = 99) achieves an end-off  
23 shift to the right by 2 positions with the boundary value of 99 and has the value  
24 [99, 99, 1, 2, 3, 4].

25 *Case (ii):* The rows of an array of rank two may all be shifted by the same amount or by  
26 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, BOUND-  
 ARY = '\*', 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.

**Example.** EPSILON (X) has the value  $2^{-23}$  for real X whose model is as in Note 13.5.

### 13.7.60 ERF (X)

**Description.** Error function.

**Class.** Elemental function.

**Argument.** X shall be of type real.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to the error function of X,  $\frac{2}{\pi} \int_0^X \exp -t^2 dt$ .

**Example.** ERF (1.0) has the value 0.843 (approximately).

### 13.7.61 ERFC (X)

**Description.** Complementary error function.

**Class.** Elemental function.

**Argument.** X shall be of type real.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to the complementary error function (that is,  $1 - \text{ERF}(X)$ ) of X.

**Example.** ERFC (1.0) has the value 0.157 (approximately).

### 13.7.62 ERFC\_SCALED (X)

- 1       **Description.** Exponentially-scaled complementary error function.
- 2       **Class.** Elemental function.
- 3       **Argument.** X shall be of type real.
- 4       **Result Characteristics.** Same as X.
- 5       **Result Value.** The result has a value equal to a processor-dependent approximation to the
- 6       exponentially-scaled complementary error function,  $\exp(X^2) \frac{2}{\sqrt{\pi}} \int_X^\infty \exp(-t^2) dt$ .
- 7       **Example.** ERFC\_SCALED (20.0) has the value 0.02817434874 (approximately).

**NOTE 13.10**

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  $X > \text{HUGE}(X)/\sqrt{\pi}$ .

### 13.7.63 EXECUTE\_COMMAND\_LINE (COMMAND [, WAIT, EXITSTAT, CMDSTAT, CMDMSG ])

- 9       **Description.** Execute the command line specified by the string COMMAND. If the processor
- 10       supports command line execution, it shall support synchronous and may support asynchronous
- 11       execution of the command line.
- 12       **Class.** Subroutine.
- 13       **Arguments.**
- |                     |                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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.                                                                                                                                                                                                                                                             |
| 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.                                                                                                                                                             |
| EXITSTAT (optional) | 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.                                                                                                                                                                                         |
| CMDSTAT (optional)  | 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. |

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. When the command is executed synchronously, EXECUTE\_COMMAND\_LINE returns after the command line has completed execution. Otherwise, EXECUTE\_COMMAND\_LINE returns without waiting.

If an error condition occurs and CMDSTAT is not present, execution of the program is terminated.

### 13.7.64 EXP (X)

**Description.** Exponential.

**Class.** Elemental function.

**Argument.** X shall be of type real or complex.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to  $e^X$ . If X is of type complex, its imaginary part is regarded as a value in radians.

**Example.** EXP (1.0) has the value 2.7182818 (approximately).

### 13.7.65 EXPONENT (X)

**Description.** Returns the exponent part of the argument when represented as a model number.

**Class.** Elemental function.

**Argument.** X shall be of type real.

**Result Characteristics.** Default integer.

**Result Value.** The result has a value equal to the exponent  $e$  of the representation for the value of X in the model (13.4) that has the radix of X but no limits on exponent values, provided X is nonzero and  $e$  is within the range for default integers. If X has the value zero, the result has the value zero. If X is an IEEE infinity or NaN, the result has the value HUGE(0).

**Examples.** EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for reals whose model is as in Note 13.5.

### 13.7.66 EXTENDS\_TYPE\_OF (A, MOLD)

**Description.** Inquires whether the dynamic type of A is an extension type (4.5.7) of the dynamic type of MOLD.

**Class.** Inquiry function.

**Arguments.**

A shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

MOLD shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

**Result Characteristics.** Default logical scalar.

**Result Value.** If MOLD is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable, the result is true; otherwise if A is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable, the result is false; otherwise the result is true if and only if the dynamic type of A is an extension type of the dynamic type of MOLD.

**NOTE 13.11**

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

### 13.7.67 FINDLOC (ARRAY, VALUE, DIM, [,MASK, KIND, BACK]) or FINDLOC (ARRAY, VALUE, [, MASK, KIND, BACK])

**Description.** Determine the location of the first element of ARRAY identified by MASK along dimension DIM having a value equal to VALUE.

**Class.** Transformational function.

**Arguments.**

ARRAY shall be of intrinsic type. It shall be an array.

VALUE shall be in type conformance with ARRAY, as specified in Table 7.10.

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.

NOTE TO THE EDITOR: type set the d1, ei, sDIM+1, etc. as in MAXLOC

**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 FINDLOC (ARRAY, VALUE) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value matches VALUE. The  $i^{\text{th}}$  subscript returned lies in the range 1 to  $e_i$ , where  $e_i$  is the extent of the  $i^{\text{th}}$  dimension of ARRAY. If no elements match VALUE or ARRAY has size zero, all elements of the result are zero.

*Case (ii):* The result of FINDLOC (ARRAY, VALUE, MASK = MASK) is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value matches VALUE. The  $i^{\text{th}}$  subscript returned lies in the range 1 to  $e_i$ , where  $e_i$  is the extent of the  $i^{\text{th}}$  dimension of ARRAY. If no elements match VALUE, ARRAY has size zero, or every element of MASK has the value false, all elements of the result are zero.

*Case (iii):* If ARRAY has rank one,  
FINDLOC (ARRAY, VALUE, DIM=DIM [, MASK = MASK]) is a scalar whose value is equal to that of the first element of FINDLOC (ARRAY, VALUE [,

MASK = MASK]). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of the result is equal to FINDLOC (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ , VALUE, DIM=1, [, MASK = MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ )).

If both ARRAY and VALUE are of type logical, the comparison is performed as array-element .EQV. VALUE; otherwise, the comparison is performed as array-element == VALUE. If the value of the comparison is true, array-element matches VALUE.

If only one element matches VALUE, that element's subscripts are returned. Otherwise, if more than one element matches 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.

#### Examples.

*Case (i):* The value of FINDLOC ((/2, 6, 4, 6 /), 6) is [2], and the value of FINDLOC ((/2, 6, 4, 6 /), 6, BACK=.TRUE.) is [4].

*Case (ii):* If A has the value  $\begin{bmatrix} 0 & -5 & 7 & 7 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & 7 \end{bmatrix}$ , and M has the value  $\begin{bmatrix} T & T & F & T \\ T & T & F & T \\ T & T & F & T \end{bmatrix}$ , FINDLOC (A, 7, MASK = M) has the value [1, 4] and FINDLOC (A, 7, MASK = M, BACK = .TRUE.) has the value [3, 4]. Note that this is independent of the declared lower bounds for A.

*Case (iii):* The value of FINDLOC ([2, 6, 4], VALUE=6, DIM=1) is 2. If B has the value  $\begin{bmatrix} 1 & 2 & -9 \\ 2 & 2 & 6 \end{bmatrix}$ , FINDLOC (B, 2, DIM=1) has the value [2, 1, 0] and FINDLOC (B, 2, DIM=2) has the value [2, 1]. Note that this is true even if B has a declared lower bound other than 1.

### 13.7.68 FLOOR (A [, KIND])

**Description.** Returns the greatest integer less than or equal to its argument.

**Class.** Elemental function.

**Arguments.**

A shall be of type real.

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.

**Result Value.** The result has a value equal to the greatest integer less than or equal to A.

**Examples.** FLOOR (3.7) has the value 3. FLOOR (-3.7) has the value -4.

### 13.7.69 FORM\_TEAM (TEAM, IMAGES)

**Description.** Form a team of images.

**Class.** Collective subroutine.

**J3 internal note**

Unresolved Technical Issue 63

I don't see why `FORM_TEAM` needs synchronisation – that seems completely unnecessary (it has no co-array argument, no need for communication – its actual arguments are all simple local variables).

If this needs synchronisation, why does `SYNC_IMAGES` not need synchronisation?

1       **Arguments.**

2           TEAM           shall be a scalar of type `IMAGE_TEAM` (13.8.3.5). It is an `INTENT(OUT)` argument.

3           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 have the same value 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. It shall not have size zero.

4       **Example.**

5       The following splits images into two equal groups and implicitly synchronizes each of the teams  
6       if there are two or more images. If there is only one image, that image becomes the only team  
7       member.

```
8  USE, INTRINSIC :: ISO_FORTRAN_ENV
9  INTEGER :: I
10 TYPE(IMAGE_TEAM) :: TEAM
11 IF (THIS_IMAGE() <= NUM_IMAGES()/2) THEN
12   CALL FORM_TEAM(TEAM, [(I, I=1, NUM_IMAGES()/2)])
13 ELSE
14   CALL FORM_TEAM(TEAM, [(I, I=NUM_IMAGES()/2+1, NUM_IMAGES())])
15 END IF
```

16   **13.7.70   FRACTION (X)**

17       **Description.** Returns the fractional part of the model representation of the argument value.

18       **Class.** Elemental function.

19       **Argument.** X shall be of type real.

20       **Result Characteristics.** Same as X.

21       **Result Value.** The result has the value  $X \times b^{-e}$ , where  $b$  and  $e$  are as defined in 13.4 for the  
22       model representation of X. If X has the value zero, the result has the value zero. If X is an IEEE  
23       infinity, the result is that infinity. If X is an IEEE NaN, the result is that NaN.

24       **Example.** `FRACTION (3.0)` has the value 0.75 for reals whose model is as in Note 13.5.

25   **13.7.71   GAMMA (X)**

26       **Description.** Gamma function.

27       **Class.** Elemental function.



1       **Argument.** X shall be of type real. Its value shall not be a negative integer or zero.

2       **Result Characteristics.** Same as X.

3       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
4       gamma function of X,  $\Gamma(X) = \int_0^\infty \exp -tt^{X-1} dt$ .

5       **Example.** GAMMA (1.0) has the value 1.000 (approximately).

### 6   **13.7.72   GET\_COMMAND ([COMMAND, LENGTH, STATUS])**

7       **Description.** Returns the entire command by which the program was invoked.

8       **Class.** Subroutine.

9       **Arguments.**

10       COMMAND       shall be scalar and of type default character. It is an INTENT(OUT) argu-  
      (optional)       ment. It is assigned the entire command by which the program was invoked.  
                          If the command cannot be determined, COMMAND is assigned all blanks.

11       LENGTH       shall be scalar and of type default integer. It is an INTENT(OUT) argument.  
      (optional)       It is assigned the significant length of the command by which the program  
                          was invoked. The significant length may include trailing blanks if the pro-  
                          cessor 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.

12       STATUS        shall be scalar and of type default integer. It is an INTENT(OUT) argument.  
      (optional)       It is assigned the value -1 if the COMMAND argument is present and has  
                          a length less than the significant length of the command. It is assigned a  
                          processor-dependent positive value if the command retrieval fails. Otherwise  
                          it is assigned the value 0.

### 13   **13.7.73   GET\_COMMAND\_ARGUMENT (NUMBER [, VALUE, LENGTH, STA-                   TUS])**

14       **Description.** Returns a command argument.

15       **Class.** Subroutine.

16       **Arguments.**

17       NUMBER        shall be scalar and of type default integer. It is an INTENT(IN) argument.  
  
18                       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).

19                       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.

- 1                   The remaining command arguments are numbered consecutively from 1 to the argument count in an order determined by the processor.
- 2           VALUE           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.
- 3           LENGTH       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.
- 4           STATUS       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 13.12**

One possible reason for failure is that NUMBER is negative or greater than COMMAND\_ARGUMENT\_COUNT().

5           **Example.**

```

6           Program echo
7           integer :: i
8           character :: command*32, arg*128
9           call get_command_argument(0, command)
10          write (*,*) "Command name is: ", command
11          do i = 1 , command_argument_count()
12           call get_command_argument(i, arg)
13           write (*,*) "Argument ", i, " is ", arg
14          end do
15          end program echo

```

16   **13.7.74 GET\_ENVIRONMENT\_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM\_NAME])**

17   **Description.** Gets the value of an environment variable.

18   **Class.** Subroutine.

19   **Arguments.**

20   NAME           shall be scalar and of type default character. It is an INTENT(IN) argument. The interpretation of case is processor dependent

1	VALUE (optional)	shall be a scalar of type default character. It is an INTENT(OUT) argument. It is assigned the value of the environment variable specified by NAME. VALUE is assigned all blanks if the environment variable does not exist or does not have a value or if the processor does not support environment variables.
2	LENGTH (optional)	shall be a scalar of type default integer. It is an INTENT(OUT) argument. If the specified environment variable exists and has a value, LENGTH is set to the length of that value. Otherwise LENGTH is set to 0.
3	STATUS (optional)	shall be scalar and of type default integer. It is an INTENT(OUT) argument. If the 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 greater than 2 may be returned for other error conditions.
4	TRIM_NAME (optional)	shall be a scalar of type logical. It is an INTENT(IN) argument. If TRIM_NAME is present with the value false then trailing blanks in NAME are considered significant if the processor supports trailing blanks in environment variable names. Otherwise trailing blanks in NAME are not considered part of the environment variable's name.

### 13.7.75 HUGE (X)

**Description.** Returns the largest number 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 integer, real, or bits. It may be a scalar or an array.

**Result Characteristics.** Scalar of the same type and kind type parameter as X.

**Result Value.** The result has the value  $r^q - 1$  if X is of type integer and  $(1 - b^{-p})b^{e_{\max}}$  if X is of type real, where  $r$ ,  $q$ ,  $b$ ,  $p$ , and  $e_{\max}$  are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X. If X is of type bits, the result value has all of its bits set to 1.

**Example.** HUGE (X) has the value  $(1 - 2^{-24}) \times 2^{127}$  for real X whose model is as in Note 13.5.

### 13.7.76 HYPOT (X, Y)

**Description.** Euclidean distance function.

**Class.** Elemental function.

**Arguments.**

X shall be of type real.

Y shall be of type real with the same kind type parameter as X.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to the

1 Euclidean distance,  $\sqrt{X^2 + Y^2}$ , without undue overflow or underflow.

2 **Example.** HYPOT (2.0, 1.0) has the value 2.236 (approximately).

### 3 13.7.77 IACHAR (C [, KIND])

4 **Description.** Returns the position of a character in the ASCII collating sequence. This is the  
5 inverse of the ACHAR function.

6 **Class.** Elemental function.

7 **Arguments.**

8 C shall be of type character and of length one.

9 KIND (optional) shall be a scalar integer initialization expression.

10 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
11 by the value of KIND; otherwise, the kind type parameter is that of default integer type.

12 **Result Value.** If C is in the collating sequence defined by the codes specified in ISO/IEC  
13 646:1991 (International Reference Version), the result is the position of C in that sequence and  
14 satisfies the inequality  $(0 \leq \text{IACHAR}(C) \leq 127)$ . A processor-dependent value is returned if C  
15 is not in the ASCII collating sequence. The results are consistent with the LGE, LGT, LLE,  
16 and LLT lexical comparison functions. For example, if LLE (C, D) is true,  $\text{IACHAR}(C) \leq$   
17  $\text{IACHAR}(D)$  is true where C and D are any two characters representable by the processor.

18 **Example.** IACHAR ('X') has the value 88.

### 19 13.7.78 IALL (ARRAY, DIM [, MASK]) or IALL (ARRAY, [MASK])

20 **Description.** Bitwise AND of all the elements of ARRAY along dimension DIM corresponding  
21 to the true elements of MASK.

22 **Class.** Transformational function.

23 **Arguments.**

24 ARRAY shall be of type integer or bits. It shall be an array.

DIM (optional) shall be a scalar of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  
25  $n$  is the rank of ARRAY. The corresponding actual argument shall not be an  
optional dummy argument.

26 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

27 **Result Characteristics.** The result is of the same type and kind type parameter as ARRAY.  
28 It is scalar if DIM is absent or if ARRAY has rank one; otherwise, the result is an array of rank  
29  $n - 1$  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  
30 ARRAY.

31 **Result Value.**

32 *Case (i):* The result of IALL (ARRAY) has a value equal to the bitwise AND of all  
33 the elements of ARRAY. If ARRAY has size zero the result value is equal to  
34 NOT (INT (0, KIND (ARRAY))) if ARRAY is of type integer, and  
35 NOT (BITS (B'0', KIND (ARRAY))) otherwise.

36 *Case (ii):* The result of IALL (ARRAY, MASK=MASK) has a value equal to

1 IALL (PACK (ARRAY, MASK)).

2 *Case (iii):* The result of IALL (ARRAY, DIM=DIM [, MASK=MASK]) has a value equal to  
 3 that of IALL (ARRAY, [, MASK=MASK]) if ARRAY has rank one. Otherwise,  
 4 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  
 5 IALL (ARRAY ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ ) [, MASK = MASK ( $s_1,$   
 6  $s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ )]).

7 **Examples.** IALL ([B'1110', B'1101', B'1011']) has the value B'1000'. IALL ([B'1110', B'1101',  
 8 B'1011'], MASK=[.true., .false., .true]) has the value B'1010'. IALL([14, 13, 11]) has the value  
 9 8. IALL([14, 13, 11], MASK=[.true., .false., .true]) has the value 10.

## 10 13.7.79 IAND (I, J)

11 **Description.** Performs a bitwise AND.

12 **Class.** Elemental function.

13 **Arguments.**

14 I shall be of type integer or bits.

15 J shall be of type integer or bits. BITS\_KIND (I) shall be equal to  
 BITS\_KIND (J).

16 **Result Characteristics.** If I and J have the same type, the result characteristics are the same  
 17 as I. Otherwise, the result characteristics are the same as those of the argument with integer  
 18 type.

19 **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
 20 the following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0
0	0	0

21 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

22 **Examples.** IAND (1, 3) has the value 1. IAND (Z'12345678', Z'0000FFFF') has the value  
 23 Z'00005678'.

## 24 13.7.80 IANY (ARRAY, DIM [, MASK]) or IANY (ARRAY, [MASK])

25 **Description.** Bitwise OR of all the elements of ARRAY along dimension DIM corresponding  
 26 to the true elements of MASK.

27 **Class.** Transformational function.

28 **Arguments.**

29 ARRAY shall be of type integer or bits. It shall be an array.

30 DIM (optional) 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 dummy argument.

1 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

2 **Result Characteristics.** The result is of the same type and kind type parameter as ARRAY.  
 3 It is scalar if DIM is absent or if ARRAY has rank one; otherwise, the result is an array of rank  
 4  $n - 1$  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  
 5 ARRAY.

6 **Result Value.**

7 *Case (i):* The result of IANY (ARRAY) is the bitwise OR of all the elements of ARRAY.  
 8 If ARRAY has size zero the result value is equal to zero if ARRAY is of type  
 9 integer, and BITS (B'0', KIND(ARRAY)) otherwise.

10 *Case (ii):* The result of IANY (ARRAY, MASK=MASK) has a value equal to  
 11 IANY (PACK (ARRAY, MASK)).

12 *Case (iii):* The result of IANY (ARRAY, DIM=DIM [, MASK=MASK]) has a value equal to  
 13 that of IANY (ARRAY, [, MASK=MASK]) if ARRAY has rank one. Otherwise,  
 14 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  
 15 IANY (ARRAY ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ ) [, MASK = MASK( $s_1,$   
 16  $s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ )]).

17 **Examples.** IANY ([B'1110', B'1101', B'1000']) has the value B'1111'. IANY ([B'1110', B'1101',  
 18 B'1000'], MASK=[.true., .false., .true]) has the value B'1110'. IANY ([14, 13, 8]) has the value  
 19 15. IANY ([14, 13, 8], MASK=[.true., .false., .true]) has the value 14.

## 20 13.7.81 IBCLR (I, POS)

21 **Description.** Clears one bit to zero.

22 **Class.** Elemental function.

23 **Arguments.**

24 I shall be of type integer or bits.

25 POS shall be of type integer. It shall be nonnegative and less than BIT.SIZE (I).

26 **Result Characteristics.** Same as I.

27 **Result Value.** The result has the value of the sequence of bits of I, except that bit POS is  
 28 zero. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

29 **Examples.** IBCLR (14, 1) has the value 12. If V has the value [1, 2, 3, 4], the value of  
 30 IBCLR (POS = V, I = 31) is [29, 27, 23, 15]. IBCLR (B'11111', 3) has the value B'10111'.

## 31 13.7.82 IBITS (I, POS, LEN)

32 **Description.** Extracts a sequence of bits.

33 **Class.** Elemental function.

34 **Arguments.**

35 I shall be of type integer or bits.

36 POS shall be of type integer. It shall be nonnegative and POS + LEN shall be  
 37 less than or equal to BIT.SIZE (I).

37 LEN shall be of type integer and nonnegative.

1       **Result Characteristics.** Same as I.

2       **Result Value.** The result has the value of the sequence of LEN bits in I beginning at bit POS,  
3       right-adjusted and with all other bits zero. The model for the interpretation of an integer value  
4       as a sequence of bits is in 13.3.

5       **Examples.** IBITS (14, 1, 3) has the value 7. IBITS (Z'ABCD', 4, 8) has the value Z'00BC'.

### 6   **13.7.83    IBSET (I, POS)**

7       **Description.** Sets one bit to one.

8       **Class.** Elemental function.

9       **Arguments.**

10      I                   shall be of type integer or bits.

11      POS                shall be of type integer. It shall be nonnegative and less than BIT\_SIZE (I).

12      **Result Characteristics.** Same as I.

13      **Result Value.** The result has the value of the sequence of bits of I, except that bit POS is one.  
14      The model for the interpretation of an integer value as a sequence of bits is in 13.3.

15      **Examples.** IBSET (12, 1) has the value 14. If V has the value [1, 2, 3, 4], the value of  
16      IBSET (POS = V, I = 0) is [2, 4, 8, 16]. IBSET (B'00000', 3) has the value B'01000'.

### 17   **13.7.84    ICHAR (C [, KIND])**

18      **Description.** Returns the position of a character in the processor collating sequence associated  
19      with the kind type parameter of the character. This is the inverse of the CHAR function.

20      **Class.** Elemental function.

21      **Arguments.**

22      C                   shall be of type character and of length one. Its value shall be that of a  
                            character capable of representation in the processor.

23      KIND (optional) shall be a scalar integer initialization expression.

24      **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
25      by the value of KIND; otherwise, the kind type parameter is that of default integer type.

26      **Result Value.** The result is the position of C in the processor collating sequence associated  
27      with the kind type parameter of C and is in the range  $0 \leq \text{ICHAR}(C) \leq n - 1$ , where  $n$  is  
28      the number of characters in the collating sequence. For any characters C and D capable of  
29      representation in the processor,  $C \leq D$  is true if and only if  $\text{ICHAR}(C) \leq \text{ICHAR}(D)$  is true  
30      and  $C == D$  is true if and only if  $\text{ICHAR}(C) == \text{ICHAR}(D)$  is true.

31      **Example.** ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence for  
32      the default character type.

### 33   **13.7.85    IEOR (I, J)**

34      **Description.** Performs a bitwise exclusive OR.

35      **Class.** Elemental function.

1       **Arguments.**

2       I               shall be of type integer or bits.

3       J               shall be of type integer or bits.   BITS\_KIND (I) shall be equal to  
BITS\_KIND (J).4       **Result Characteristics.** If I and J have the same type, the result characteristics are the same  
5       as I. Otherwise, the result characteristics are the same as those of the argument with integer  
6       type.7       **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
8       the following truth table:

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

9       The model for the interpretation of an integer value as a sequence of bits is in 13.3.

10       **Examples.** IEOR (1, 3) has the value 2. IEOR (Z'12340000', Z'1234FFFF') has the value  
11       Z'0000FFFF'.12   **13.7.86   IMAGE\_INDEX (CO\_ARRAY, SUB)**13       **Description.** Returns the index of the image corresponding to the co-subscripts SUB for  
14       CO\_ARRAY.15       **Class.** Inquiry function.16       **Arguments.**

17       CO\_ARRAY       shall be a co-array of any type.

18       SUB            shall be a rank-one array of size equal to the co-rank of CO\_ARRAY and  
shall be of type integer.19       **Result Characteristics.** Default integer scalar.20       **Result Value.** If the value of SUB is a valid sequence of co-subscripts for CO\_ARRAY, the  
21       result is the index of the corresponding image. Otherwise, the result is zero.22       **Examples.** If A is declared A [0:\*], IMAGE\_INDEX (A, (/0/)) has the value 1. If B is declared  
23       REAL B (10, 20) [10, 0:9, 0:\*], IMAGE\_INDEX (B, [3, 1, 2]) has the value 213 (on any image).**NOTE 13.13**

For an example of a module that implements a function similar to the intrinsic THIS\_IMAGE, see subclause C.10.1.

24   **13.7.87   INDEX (STRING, SUBSTRING [, BACK, KIND])**25       **Description.** Returns the starting position of a substring within a string.26       **Class.** Elemental function.



**Arguments.**

STRING shall be of type character.

SUBSTRING shall be of type character with the same kind type parameter as STRING.

BACK (optional) shall be of type logical.

KIND (optional) shall be a scalar integer 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.

**Result Value.**

*Case (i):* If BACK is absent or has the value false, the result is the minimum positive value of  $I$  such that  $STRING(I : I + LEN(SUBSTRING) - 1) = SUBSTRING$  or zero if there is no such value. Zero is returned if  $LEN(STRING) < LEN(SUBSTRING)$  and one is returned if  $LEN(SUBSTRING) = 0$ .

*Case (ii):* If BACK is present with the value true, the result is the maximum value of  $I$  less than or equal to  $LEN(STRING) - LEN(SUBSTRING) + 1$  such that  $STRING(I : I + LEN(SUBSTRING) - 1) = SUBSTRING$  or zero if there is no such value. Zero is returned if  $LEN(STRING) < LEN(SUBSTRING)$  and  $LEN(STRING) + 1$  is returned if  $LEN(SUBSTRING) = 0$ .

**Examples.** INDEX('FORTRAN', 'R') has the value 3.

INDEX('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

**13.7.88 INT (A [, KIND])**

**Description.** Convert to integer type.

**Class.** Elemental function.

**Arguments.**

A shall be of type integer, real, complex, or bits.

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.

**Result Value.**

*Case (i):* If A is of type integer,  $INT(A) = A$ .

*Case (ii):* If A is of type real, there are two cases: if  $|A| < 1$ ,  $INT(A)$  has the value 0; if  $|A| \geq 1$ ,  $INT(A)$  is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

*Case (iii):* If A is of type complex,  $INT(A) = INT(REAL(A, KIND(A)))$ .

*Case (iv):* If A is of type bits, the result has the integer value such that  $BITS(INT(A, kind)) = BITS(A, BITS.KIND(INT(0, kind)))$ , where *kind* is the kind of the result. If the value specified by the model in 13.3 for interpreting the bits as an integer is a valid value for the result, it has that value.

**Example.** INT(-3.7) has the value -3.

**13.7.89 IOR (I, J)**

1       **Description.** Performs a bitwise inclusive OR.

2       **Class.** Elemental function.

3       **Arguments.**

4       I                   shall be of type integer or bits.

5       J                   shall be of type integer or bits.   BITS\_KIND (I) shall be equal to  
BITS\_KIND (J).

6       **Result Characteristics.** If I and J have the same type, the result characteristics are the same  
7       as I. Otherwise, the result characteristics are the same as those of the argument with integer  
8       type.

9       **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
10      the following truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

11      The model for the interpretation of an integer value as a sequence of bits is in 13.3.

12      **Examples.** IOR (5, 3) has the value 7. IOR (Z'12345678', Z'0000FFFF') has the value  
13      Z'1234FFFF'.

### 14   **13.7.90   IPARITY (ARRAY, DIM [, MASK]) or IPARITY (ARRAY, [MASK])**

15      **Description.** Bitwise exclusive OR of all the elements of ARRAY along dimension DIM corre-  
16      sponding to the true elements of MASK.

17      **Class.** Transformational function.

18      **Arguments.**

19      ARRAY              shall be of type integer or bits. It shall be an array.

20      DIM (optional)    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 dummy argument.

21      MASK (optional) shall be of type logical and shall be conformable with ARRAY.

22      **Result Characteristics.** The result is of the same type and kind type parameter as AR-  
23      RAY. It is scalar if DIM is absent; otherwise, the result has rank  $n - 1$  and shape  $(d_1, d_2,$   
24       $\dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

25      **Result Value.**

26      *Case (i):*       The result of IPARITY (ARRAY) has a value equal to the bitwise exclusive OR  
27      of all the elements of ARRAY. If ARRAY has size zero the result has the value  
28      zero if ARRAY is of type integer, and BITS (B'0', KIND (ARRAY)) otherwise.

29      *Case (ii):*     The result of IPARITY (ARRAY, MASK=MASK) has a value equal to that of  
30      IPARITY (PACK (ARRAY, MASK)).

1        *Case (iii):*    The result of IPARITY (ARRAY, DIM=DIM [, MASK=MASK]) has a value  
 2        equal to that of IPARITY (ARRAY, [, MASK=MASK]) if ARRAY has rank one.  
 3        Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of the result  
 4        is equal to IPARITY (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  [, MASK  
 5        = MASK $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ ]).

6        **Examples.** IPARITY ([B'1110', B'1101', B'1000']) has the value B'1011'.  
 7        IPARITY ([B'1110', B'1101', B'1000'], MASK=[.true., .false., .true]) has the value B'0110'.  
 8        IPARITY ([14, 13, 8]) has the value 11. IPARITY ([14, 13, 8], MASK=[.true., .false., .true])  
 9        has the value 6.

### 10    13.7.91    ISHFT (I, SHIFT)

11        **Description.** Performs a logical shift.

12        **Class.** Elemental function.

13        **Arguments.**

14        I                    shall be of type integer or bits.

15        SHIFT               shall be of type integer. The absolute value of SHIFT shall be less than or  
                                  equal to BIT\_SIZE (I).

16        **Result Characteristics.** Same as I.

17        **Result Value.** The result has the value obtained by shifting the bits of I by SHIFT positions.  
 18        If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right;  
 19        and if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as  
 20        appropriate, are lost. Zeros are shifted in from the opposite end. The model for the interpretation  
 21        of an integer value as a sequence of bits is in 13.3.

22        **Examples.** ISHFT (3, 1) has the value 6. ISHFT (B'00000011', 1) has the value B'00000110'.

### 23    13.7.92    ISHFTC (I, SHIFT [, SIZE])

24        **Description.** Performs a circular shift of the rightmost bits.

25        **Class.** Elemental function.

26        **Arguments.**

27        I                    shall be of type integer or bits.

28        SHIFT               shall be of type integer. The absolute value of SHIFT shall be less than or  
                                  equal to SIZE.

29        SIZE (optional)    shall be of type integer. The value of SIZE shall be positive and shall not  
                                  exceed BIT\_SIZE (I). If SIZE is absent, it is as if it were present with the  
                                  value of BIT\_SIZE (I).

30        **Result Characteristics.** Same as I.

31        **Result Value.** The result has the value obtained by shifting the SIZE rightmost bits of I  
 32        circularly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative,  
 33        the shift is to the right; and if SHIFT is zero, no shift is performed. No bits are lost. The  
 34        unshifted bits are unaltered. The model for the interpretation of an integer value as a sequence  
 35        of bits is in 13.3.

1       **Examples.** ISHFTC (3, 2, 3) has the value 5. ISHFTC (Z'ABCD', 4, 8) has the value Z'ABDC'.

## 2   **13.7.93   IS\_CONTIGUOUS (A)**

3       **Description.** Determine whether an object is contiguous (5.3.6).

4       **Class.** Inquiry function.

5       **Argument.** A may be of any type. It shall be an assumed-shape array or an array pointer. If  
6 it is a pointer it shall be associated.

7       **Result Characteristics.** Default logical scalar.

8       **Result Value.** The result has the value true if A is contiguous, and false otherwise.

9       **Example.** After the pointer assignment AP => TARGET (1:10:2), IS\_CONTIGUOUS (AP)  
10 will have the value false.

## 11   **13.7.94   IS\_IOSTAT\_END (I)**

12       **Description.** Determine whether a value indicates an end-of-file condition.

13       **Class.** Elemental function.

14       **Argument.** I shall be of type integer.

15       **Result Characteristics.** Default logical.

16       **Result Value.** The result has the value true if and only if I is a value for the *scalar-int-variable*  
17 in an IOSTAT= specifier (9.10.4) that would indicate an end-of-file condition.

## 18   **13.7.95   IS\_IOSTAT\_EOR (I)**

19       **Description.** Determine whether a value indicates an end-of-record condition.

20       **Class.** Elemental function.

21       **Argument.** I shall be of type integer.

22       **Result Characteristics.** Default logical.

23       **Result Value.** The result has the value true if and only if I is a value for the *scalar-int-variable*  
24 in an IOSTAT= specifier (9.10.4) that would indicate an end-of-record condition.

## 25   **13.7.96   KIND (X)**

26       **Description.** Returns the value of the kind type parameter of X.

27       **Class.** Inquiry function.

28       **Argument.** X may be of any intrinsic type. It may be a scalar or an array.

29       **Result Characteristics.** Default integer scalar.

30       **Result Value.** The result has a value equal to the kind type parameter value of X.

31       **Example.** KIND (0.0) has the kind type parameter value of default real.

## 32   **13.7.97   LBOUND (ARRAY [, DIM, KIND])**

**Description.** Returns all the lower bounds or a specified lower bound of an array.

**Class.** Inquiry function.

**Arguments.**

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.

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.

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 rank of ARRAY.

**Result Value.**

*Case (i):*        If ARRAY is a whole array or array structure component and either ARRAY is an assumed-size array of rank DIM or dimension DIM of ARRAY has nonzero extent, LBOUND (ARRAY, DIM) has a value equal to the lower bound for subscript DIM of ARRAY. Otherwise the result value is 1.

*Case (ii):*       LBOUND (ARRAY) has a value whose  $i$ th element is equal to LBOUND (ARRAY,  $i$ ), for  $i = 1, 2, \dots, n$ , where  $n$  is the rank of ARRAY.

**Examples.** If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

### 13.7.98 LEADZ (I [, KIND])

**Description.** Count the number of leading zero bits.

**Class.** Elemental function.

**Arguments.**

I                shall be of type integer or bits.

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.

#### J3 internal note

##### Unresolved Technical Issue 65

This is another spurious and ineffective useless KIND argument. Lots of these abound in the “BITS” feature.

**Result Value.** If all of the bits of I are zero, the result has the value BIT.SIZE (I). Otherwise, the result has the value  $\text{BIT.SIZE (I)} - 1 - k$ , where  $k$  is the position of the leftmost 1 bit in I. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

1       **Examples.** LEADZ (B'001101000') has the value 2. LEADZ (1) has the value 31 if BIT\_SIZE (1)  
2       has the value 32.

### 3   **13.7.99    LEN (STRING [, KIND])**

4       **Description.** Returns the length of a character entity.

5       **Class.** Inquiry function.

6       **Arguments.**

7       STRING           shall be of type character. It may be a scalar or an array. If it is an unallocated  
                          allocatable or a pointer that is not associated, its length type parameter shall  
                          not be deferred.

8       KIND (optional) shall be a scalar integer initialization expression.

9       **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that  
10      specified by the value of KIND; otherwise the kind type parameter is that of default integer  
11      type.

12      **Result Value.** The result has a value equal to the number of characters in STRING if it is  
13      scalar or in an element of STRING if it is an array.

14      **Example.** If C is declared by the statement

15            CHARACTER (11) C (100)

16      LEN (C) has the value 11.

### 17   **13.7.100   LEN\_TRIM (STRING [, KIND])**

18      **Description.** Returns the length of the character argument without counting trailing blank  
19      characters.

20      **Class.** Elemental function.

21      **Arguments.**

22      STRING           shall be of type character.

23      KIND (optional) shall be a scalar integer initialization expression.

24      **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
25      by the value of KIND; otherwise the kind type parameter is that of default integer type.

26      **Result Value.** The result has a value equal to the number of characters remaining after any  
27      trailing blanks in STRING are removed. If the argument contains no nonblank characters, the  
28      result is zero.

29      **Examples.** LEN\_TRIM (' A B ') has the value 4 and LEN\_TRIM (' ') has the value 0.

### 30   **13.7.101   LGE (STRING\_A, STRING\_B)**

31      **Description.** Test whether a string is lexically greater than or equal to another string, based  
32      on the ASCII collating sequence.

33      **Class.** Elemental function.

1       **Arguments.**

2       STRING\_A       shall be of type default character of type ASCII character.

3       STRING\_B       shall be of type character with the same kind type parameter as STRING\_A.

4       **Result Characteristics.** Default logical.

5       **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
 6       string were extended on the right with blanks to the length of the longer string. If either string  
 7       contains a character not in the ASCII character set, the result is processor dependent. The  
 8       result is true if the strings are equal or if STRING\_A follows STRING\_B in the ASCII collating  
 9       sequence; otherwise, the result is false.

**NOTE 13.14**

The result is true if both STRING\_A and STRING\_B are of zero length.

10       **Example.** LGE ('ONE', 'TWO') has the value false.11   **13.7.102 LGT (STRING\_A, STRING\_B)**

12       **Description.** Test whether a string is lexically greater than another string, based on the ASCII  
 13       collating sequence.

14       **Class.** Elemental function.15       **Arguments.**

16       STRING\_A       shall be of type default character of type ASCII character.

17       STRING\_B       shall be of type character with the same kind type parameter as STRING\_A.

18       **Result Characteristics.** Default logical.

19       **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
 20       string were extended on the right with blanks to the length of the longer string. If either string  
 21       contains a character not in the ASCII character set, the result is processor dependent. The  
 22       result is true if STRING\_A follows STRING\_B in the ASCII collating sequence; otherwise, the  
 23       result is false.

**NOTE 13.15**

The result is false if both STRING\_A and STRING\_B are of zero length.

24       **Example.** LGT ('ONE', 'TWO') has the value false.25   **13.7.103 LLE (STRING\_A, STRING\_B)**

26       **Description.** Test whether a string is lexically less than or equal to another string, based on  
 27       the ASCII collating sequence.

28       **Class.** Elemental function.29       **Arguments.**

30       STRING\_A       shall be of type default character of type ASCII character.

31       STRING\_B       shall be of type character with the same kind type parameter as STRING\_A.

**Result Characteristics.** Default logical.

**Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if the strings are equal or if `STRING_A` precedes `STRING_B` in the ASCII collating sequence; otherwise, the result is false.

**NOTE 13.16**

The result is true if both `STRING_A` and `STRING_B` are of zero length.

**Example.** `LLE ('ONE', 'TWO')` has the value true.

### 13.7.104 LLT (STRING\_A, STRING\_B)

**Description.** Test whether a string is lexically less than another string, based on the ASCII collating sequence.

**Class.** Elemental function.

**Arguments.**

`STRING_A` shall be of type default character of type ASCII character.

`STRING_B` shall be of type character with the same kind type parameter as `STRING_A`.

**Result Characteristics.** Default logical.

**Result Value.** If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string. If either string contains a character not in the ASCII character set, the result is processor dependent. The result is true if `STRING_A` precedes `STRING_B` in the ASCII collating sequence; otherwise, the result is false.

**NOTE 13.17**

The result is false if both `STRING_A` and `STRING_B` are of zero length.

**Example.** `LLT ('ONE', 'TWO')` has the value true.

### 13.7.105 LOG (X)

**Description.** Natural logarithm.

**Class.** Elemental function.

**Argument.** `X` shall be of type real or complex. If `X` is real, its value shall be greater than zero. If `X` is complex, its value shall not be zero.

**Result Characteristics.** Same as `X`.

**Result Value.** The result has a value equal to a processor-dependent approximation to  $\log_e X$ . A result of type complex is the principal value with imaginary part  $\omega$  in the range  $-\pi \leq \omega \leq \pi$ . If the real part of `X` is less than zero and the imaginary part of `X` is zero, then the imaginary part of the result is  $\pi$  if the imaginary part of `X` is positive real zero or the processor cannot distinguish between positive and negative real zero, and  $-\pi$  if the imaginary part of `X` is negative real zero.



1       **Example.** LOG (10.0) has the value 2.3025851 (approximately).

## 2   **13.7.106 LOG\_GAMMA (X)**

3       **Description.** Logarithm of the absolute value of the gamma function.

4       **Class.** Elemental function.

5       **Argument.** X shall be of type real. Its value shall not be a negative integer or zero.

6       **Result Characteristics.** Same as X.

7       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
8       natural logarithm of the absolute value of the gamma function of X.

9       **Example.** LOG\_GAMMA (3.0) has the value 0.693 (approximately).

## 10   **13.7.107 LOG10 (X)**

11       **Description.** Common logarithm.

12       **Class.** Elemental function.

13       **Argument.** X shall be of type real. The value of X shall be greater than zero.

14       **Result Characteristics.** Same as X.

15       **Result Value.** The result has a value equal to a processor-dependent approximation to  $\log_{10}X$ .

16       **Example.** LOG10 (10.0) has the value 1.0 (approximately).

## 17   **13.7.108 LOGICAL (L [, KIND])**

18       **Description.** Converts between kinds of logical or from bits to logical.

19       **Class.** Elemental function.

20       **Arguments.**

21       L               shall be of type logical or bits.

22       KIND (optional) shall be a scalar integer initialization expression.

23       **Result Characteristics.** Logical. If KIND is present, the kind type parameter is that specified  
24       by the value of KIND; otherwise, the kind type parameter is that of default logical.

25       **Result Value.**

26       *Case (i):*       If L is of type logical, the value is that of L.

27       *Case (ii):*      If L is of type bits and KIND (L) is greater than or equal to BITS\_KIND (*result*),  
28       the result has the value whose physical representation is the same as the rightmost  
29       bits of L.

30       *Case (iii):*     If L is of type bits and KIND (L) is less than BITS\_KIND (*result*), the rightmost  
31       KIND(L) bits of the physical representation of the result value are the same as  
32       those of L, and the remaining bits of the physical representation of the result  
33       value are zero.

**J3 internal note****Unresolved Technical Issue 66**

This might not be a value. Seriously. We don't provide semantics for physical representations that don't represent a logical value. There are certainly systems that use the fastest test, which might be all-zero for a "FALSE" test and lowest-bit-set for a "TRUE" test... which is not going to give sensible results if you whack any old set of bits into the LOGICAL.

Should we not say something about that here? Or rather, say less (don't say this returns a value...) see below.

At least with TRANSFER we give what the use can rely on, viz that converting LOGICAL to something else and back again gives you what you start with. Note that also with TRANSFER we carefully don't say that the result is a "value" of the result type, we only say what its physical representation is. That is, I think, what is needed here.

Actually, it's also probably what is needed for INT (bits) and REAL (bits) too. We get this right for CHAR already, but by a different means.

1       **Examples.** LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical,  
2       regardless of the kind type parameter of the logical variable L. LOGICAL (BITS (.TRUE.)) has  
3       the value true.

#### 4   **13.7.109   MASKL (I, [, KIND])**

5       **Description.** Generate a left justified mask.

6       **Class.** Elemental function.

7       **Arguments.**

8       I               shall be of type integer. It shall be nonnegative and less than or equal to the  
                          kind type parameter of the result.

9       KIND (optional) shall be a scalar integer initialization expression.

10      **Result Characteristics.** Bits. If KIND is present, the kind type parameter is that specified  
11      by the value of KIND; otherwise, the kind type parameter is that of default bits type.

12      **Result Value.** The result value has its leftmost I bits set to 1 and the remaining bits set to 0.

13      **Example.** MASKL (12) has the value Z'FFF00000' if the default bits kind type parameter  
14      value is 32.

#### 15   **13.7.110   MASKR (I, [, KIND])**

16      **Description.** Generate a right justified mask.

17      **Class.** Elemental function.

18      **Arguments.**

19      I               shall be of type integer. It shall be nonnegative and less than or equal to the  
                          kind type parameter of the result.

20      KIND (optional) shall be a scalar integer initialization expression.

21      **Result Characteristics.** Bits. If KIND is present, the kind type parameter is that specified  
22      by the value of KIND; otherwise, the kind type parameter is that of default bits type.

23      **Result Value.** The result value has its rightmost I bits set to 1 and the remaining bits set to

1 0.

2 **Example.** MASKR (12) has the value Z'00000FFF' if the default bits kind type parameter  
3 value is 32.

#### 4 13.7.111 MATMUL (MATRIX\_A, MATRIX\_B)

5 **Description.** Performs matrix multiplication of numeric or logical matrices.

6 **Class.** Transformational function.

7 **Arguments.**

8 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.

9 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.

10 **Result Characteristics.** If the arguments are of numeric type, the type and kind type pa-  
11 rameter of the result are determined by the types of the arguments as specified in 7.1.4.2 for  
12 the \* operator. If the arguments are of type logical, the result is of type logical with the kind  
13 type parameter of the arguments as specified in 7.1.4.2 for the .AND. operator. The shape of  
14 the result depends on the shapes of the arguments as follows:

15 *Case (i):* If MATRIX\_A has shape  $(n, m)$  and MATRIX\_B has shape  $(m, k)$ , the result has  
16 shape  $(n, k)$ .

17 *Case (ii):* If MATRIX\_A has shape  $(m)$  and MATRIX\_B has shape  $(m, k)$ , the result has  
18 shape  $(k)$ .

19 *Case (iii):* If MATRIX\_A has shape  $(n, m)$  and MATRIX\_B has shape  $(m)$ , the result has  
20 shape  $(n)$ .

21 **Result Value.**

22 *Case (i):* Element  $(i, j)$  of the result has the value SUM (MATRIX\_A  $(i, :)$  \* MATRIX\_B  $(:, j)$ ) if the arguments are of numeric type and has the value ANY (MATRIX\_A  $(i, :)$  .AND. MATRIX\_B  $(:, j)$ ) if the arguments are of logical type.

25 *Case (ii):* Element  $(j)$  of the result has the value SUM (MATRIX\_A  $(:) * MATRIX_B (:, j)$ ) if the arguments are of numeric type and has the value ANY (MATRIX\_A  $(:) .AND. MATRIX_B (:, j)$ ) if the arguments are of logical type.

28 *Case (iii):* Element  $(i)$  of the result has the value SUM (MATRIX\_A  $(i, :)$  \* MATRIX\_B  $(:)$ ) if the arguments are of numeric type and has the value ANY (MATRIX\_A  $(i, :)$  .AND. MATRIX\_B  $(:)$ ) if the arguments are of logical type.

31 **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  
32 vectors [1, 2] and [1, 2, 3].

33 *Case (i):* The result of MATMUL (A, B) is the matrix-matrix product AB with the value  
34  $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$ .

1       *Case (ii):*     The result of MATMUL (X, A) is the vector-matrix product XA with the value  
2                     [5, 8, 11].

3       *Case (iii):*   The result of MATMUL (A, Y) is the matrix-vector product AY with the value  
4                     [14, 20].

### 5   **13.7.112   MAX (A1, A2 [, A3, ...])**

6       **Description.** Maximum value.

7       **Class.** Elemental function.

8       **Arguments.** The arguments shall all have the same type which shall be integer, real, bits, or  
9                     character and they shall all have the same kind type parameter.

10      **Result Characteristics.** The type and kind type parameter of the result are the same as those  
11                     of the arguments. For arguments of character type, the length of the result is the length of the  
12                     longest argument.

13      **Result Value.** The value of the result is that of the largest argument. For arguments of  
14                     character type, the result is the value that would be selected by application of intrinsic relational  
15                     operators; that is, the collating sequence for characters with the kind type parameter of the  
16                     arguments is applied. If the selected argument is shorter than the longest argument, the result  
17                     is extended with blanks on the right to the length of the longest argument.

18      **Examples.** MAX (-9.0, 7.0, 2.0) has the value 7.0, MAX ('Z', 'BB') has the value 'Z ', and  
19                     MAX ((/'A', 'Z'/), (/BB', 'Y '/)) has the value (/BB', 'Z '/).MAX (B'10000', B'01111') has  
20                     the value B'10000'.

### 21   **13.7.113   MAXEXPONENT (X)**

22      **Description.** Returns the maximum exponent of the model representing numbers of the same  
23                     type and kind type parameter as the argument.

24      **Class.** Inquiry function.

25      **Argument.** X shall be of type real. It may be a scalar or an array.

26      **Result Characteristics.** Default integer scalar.

27      **Result Value.** The result has the value  $e_{\max}$ , as defined in 13.4 for the model representing  
28                     numbers of the same type and kind type parameter as X.

29      **Example.** MAXEXPONENT (X) has the value 127 for real X whose model is as in Note 13.5.

### 30   **13.7.114   MAXLOC (ARRAY, DIM [, MASK, KIND, BACK]) or 31                     MAXLOC (ARRAY [, MASK, KIND, BACK])**

32      **Description.** Determine the location of the first element of ARRAY along dimension DIM  
33                     having the maximum value of the elements identified by MASK.

34      **Class.** Transformational function.

35      **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

$$\text{MAXLOC}(\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n), \text{DIM}=1 [, \text{MASK} = \text{MASK}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)]).$$

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.

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

#### Examples.

*Case (i):* The value of MAXLOC ((/ 2, 6, 4, 6 /)) is [2] and the value of MAXLOC ([2, 6, 4, 6], BACK=.TRUE.) is [4].

*Case (ii):* If A has the value  $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$ , MAXLOC (A, MASK = A < 6) has the value [3, 2]. Note that this is independent of the declared lower bounds for A.

*Case (iii):* The value of MAXLOC ((/ 5, -9, 3 /), DIM = 1) is 1. If B has the value

1  $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$ , MAXLOC (B, DIM = 1) is [2, 1, 2] and MAXLOC (B, DIM = 2)  
 2 is [2, 3]. Note that this is independent of the declared lower bounds for B.

### 3 13.7.115 MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])

4 **Description.** Maximum value of the elements of ARRAY along dimension DIM corresponding  
 5 to the true elements of MASK.

6 **Class.** Transformational function.

7 **Arguments.**

8 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  $1 \leq \text{DIM} \leq n$ ,  
 9 where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
 be an optional dummy argument.

10 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

11 **Result Characteristics.** The result is of the same type and type parameters as ARRAY. It is  
 12 scalar if DIM is absent; otherwise, the result has rank  $n-1$  and shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$   
 13  $\dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

14 **Result Value.**

15 *Case (i):* The result of MAXVAL (ARRAY) has a value equal to the maximum value of  
 16 all the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size  
 17 zero and type integer or real, the result has the value of the negative number of  
 18 the largest magnitude supported by the processor for numbers of the type and  
 19 kind type parameter of ARRAY. If ARRAY has size zero and type character, the  
 20 result has the value of a string of characters of length LEN (ARRAY), with each  
 21 character equal to CHAR (0, KIND = KIND (ARRAY)).

22 *Case (ii):* The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to that of  
 23 MAXVAL (PACK (ARRAY, MASK)).

24 *Case (iii):* The result of MAXVAL (ARRAY, DIM = DIM [, MASK = MASK]) has a value  
 25 equal to that of MAXVAL (ARRAY [, MASK = MASK]) if ARRAY has rank  
 26 one. Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of the  
 27 result is equal to

28  $\text{MAXVAL} (\text{ARRAY} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$   
 29  $[\text{, MASK = MASK} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)] )$ .

30 If ARRAY has type character, the result is the value that would be selected by application of  
 31 intrinsic relational operators; that is, the collating sequence for characters with the kind type  
 32 parameter of the arguments is applied.

33 **Examples.**

34 *Case (i):* The value of MAXVAL ((/ 1, 2, 3 /)) is 3.

35 *Case (ii):* MAXVAL (C, MASK = C < 0.0) finds the maximum of the negative elements of  
 36 C.

37 *Case (iii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , MAXVAL (B, DIM = 1) is [2, 4, 6] and MAX-  
 38 VAL (B, DIM = 2) is [5, 6].

### 1 13.7.116 MERGE (TSOURCE, FSOURCE, MASK)

2 **Description.** Choose alternative value according to the value of a mask.

3 **Class.** Elemental function.

4 **Arguments.**

5 TSOURCE may be of any type.

6 FSOURCE shall be of the same type and type parameters as TSOURCE.

7 MASK shall be of type logical.

8 **Result Characteristics.** Same as TSOURCE.

9 **Result Value.** The result is TSOURCE if MASK is true and FSOURCE otherwise.

10 **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  
 11 MASK is the array  $\begin{bmatrix} T & . & T \\ . & . & T \end{bmatrix}$ , where “T” represents true and “.” represents false, then  
 12 MERGE (TSOURCE, FSOURCE, MASK) is  $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$ . The value of MERGE (1.0, 0.0,  
 13 K > 0) is 1.0 for K = 5 and 0.0 for K = -2.

### 14 13.7.117 MERGE\_BITS (I, J, MASK)

15 **Description.** Merge bits under mask.

16 **Class.** Elemental function.

17 **Arguments.**

18 I shall be of type bits or integer.

19 J shall have the same type and type parameters as I.

#### J3 internal note

Unresolved Technical Issue 67

If MERGE\_BITS only handles “int,int” and “bits,bits” ... and not “int,bits” and “bits,int” ... why do IAND IOR IEOR have to handle the mixed-type cases?

This is inconsistent. Probably best fixed by removal of the mixed-type cases of IAND et al.

20 MASK shall be of type bits. Its kind type parameter shall be equal to BITS\_KIND (I).

#### J3 internal note

Unresolved Technical Issue 68

So why does this have such different requirements to J? It's not like it's being used in such a conceptually different way.

That is, why do we have “int,int,bits” and “bits,bits,bits” but not “int,int,int”?

This brings into question the whole idea of whether MERGE\_BITS should take integer at all.

21 **Result Characteristics.** Same as I.

22 **Result Value.** The result has the value of IOR (IAND (I, MASK), IAND (J, NOT (MASK))).

1       **Example.** MERGE\_BITS(Z'01234567', Z'89ABCDEF', Z'FFFF0000') has the value Z'0123CDEF'.

## 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  
6       character 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  
8       of the arguments. For arguments of character type, the length of the result is the length of the  
9       longest argument.

10       **Result Value.** The value of the result is that of the smallest argument. For arguments  
11       of character type, the result is the value that would be selected by application of intrinsic  
12       relational operators; that is, the collating sequence for characters with the kind type parameter  
13       of the arguments is applied. If the selected argument is shorter than the longest argument, the  
14       result is extended with blanks on the right 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  
17       the value B'01111'.

## 18 13.7.119   MINEXPONENT (X)

19       **Description.** Returns the minimum (most negative) exponent of the model representing num-  
20       bers of the same type and kind type parameter as the argument.

21       **Class.** Inquiry function.

22       **Argument.** X shall be of type real. It may be a scalar or an array.

23       **Result Characteristics.** Default integer scalar.

24       **Result Value.** The result has the value  $e_{\min}$ , as defined in 13.4 for the model representing  
25       numbers of the same type and kind type parameter as X.

26       **Example.** MINEXPONENT (X) has the value -126 for real X whose model is as in Note 13.5.

## 27 13.7.120   MINLOC (ARRAY, DIM [, MASK, KIND, BACK]) or MINLOC (ARRAY [, MASK, KIND, BACK])

28       **Description.** Determine the location of the first element of ARRAY along dimension DIM  
29       having the minimum value of the elements identified by MASK.

30       **Class.** Transformational function.

31       **Arguments.**

32       ARRAY           shall be of type integer, real, bits, or character. It shall be an array.

33       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  
34       be an optional dummy argument.

34       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  
4       by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM  
5       is absent, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise,  
6       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  $(d_1, d_2,$   
7        $\dots, d_n)$  is the shape of ARRAY.

8       **Result Value.**

9       *Case (i):*       The result of MINLOC (ARRAY) is a rank-one array whose element values are  
10       the values of the subscripts of an element of ARRAY whose value equals the  
11       minimum value of all the elements of ARRAY. The  $i$ th subscript returned lies  
12       in the range 1 to  $e_i$ , where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If  
13       ARRAY has size zero, all elements of the result are zero.

14       *Case (ii):*     The result of MINLOC (ARRAY, MASK = MASK) is a rank-one array whose  
15       element values are the values of the subscripts of an element of ARRAY, corre-  
16       sponding to a true element of MASK, whose value equals the minimum value of  
17       all such elements of ARRAY. The  $i$ th subscript returned lies in the range 1 to  $e_i$ ,  
18       where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If ARRAY has size zero  
19       or every element of MASK has the value false, all elements of the result are zero.

20       *Case (iii):*    If ARRAY has rank one, MINLOC (ARRAY, DIM = DIM [, MASK = MASK]) is  
21       a scalar whose value is equal to that of the first element of MINLOC (ARRAY [,  
22       MASK = MASK]). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1},$   
23        $\dots, s_n)$  of the result is equal to

24                       MINLOC (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ , DIM=1  
25                       [, MASK = MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  ] ).

26       If only one element has the minimum value, that elements subscripts are returned. Otherwise,  
27       if more than one element has the minimum value and BACK is absent or present with the value  
28       false, the element whose subscripts are returned is the first such element, taken in array element  
29       order. If BACK is present with the value true, the element whose subscripts are returned is the  
30       last such element, taken in array element order.

31       If ARRAY has type character, the result is the value that would be selected by application of  
32       intrinsic relational operators; that is, the collating sequence for characters with the kind type  
33       parameter of the arguments is applied.

34       **Examples.**

35       *Case (i):*       The value of MINLOC ((/ 4, 3, 6, 3 /)) is [2] and the value of MINLOC ([4, 3, 6, 3],  
36       BACK = .TRUE.) is [4].

37       *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  
38       the value [1, 4]. Note that this is independent of the declared lower bounds for  
39       A.

40       *Case (iii):*    The value of MINLOC ((/ 5, -9, 3 /), DIM = 1) is 2. If B has the value  
41        $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$ , MINLOC (B, DIM = 1) is [1, 2, 1] and MINLOC (B, DIM = 2)  
42       is [3, 1]. Note that this is independent of the declared lower bounds for B.

43       **13.7.121 MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])**

**Description.** Minimum value of all the elements of ARRAY along dimension DIM corresponding to true elements of 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.

**Result Characteristics.** The result is of the same type and type parameters as ARRAY. 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 ARRAY.

**Result Value.**

*Case (i):* The result of MINVAL (ARRAY) has a value equal to the minimum value of all the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size zero and type integer or real, the result has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY. If ARRAY has size zero and type character, the result has the value of a string of characters of length LEN (ARRAY), with each character equal to CHAR ( $n-1$ , KIND = KIND (ARRAY)), where  $n$  is the number of characters in the collating sequence for characters with the kind type parameter of ARRAY.

*Case (ii):* The result of MINVAL (ARRAY, MASK = MASK) has a value equal to that of MINVAL (PACK (ARRAY, MASK)).

*Case (iii):* The result of MINVAL (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of MINVAL (ARRAY [, MASK = MASK]) if ARRAY has rank one. Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of the result is equal to

$$\text{MINVAL} (\text{ARRAY} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) [, \text{MASK} = \text{MASK} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) ] ).$$

If ARRAY has type character, the result is the value that would be selected by application of intrinsic relational operators; that is, the collating sequence for characters with the kind type parameter of the arguments is applied.

**Examples.**

*Case (i):* The value of MINVAL ((/ 1, 2, 3 /)) is 1.

*Case (ii):* MINVAL (C, MASK = C > 0.0) forms the minimum of the positive elements of C.

*Case (iii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , MINVAL (B, DIM = 1) is [1, 3, 5] and MINVAL (B, DIM = 2) is [1, 2].

### 13.7.122 MOD (A, P)

**Description.** Remainder function.

**Class.** Elemental function.

1       **Arguments.**

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.  
7       MOD (-8, 5) 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  $A = Q \times P + R$ ,  
17                       where Q is an integer, the inequalities  $0 \leq R < P$  hold if  $P > 0$ , and  $P < R \leq 0$   
18                       hold if  $P < 0$ .19      *Case (ii):*     A is of type real. The value of the result is  $A - \text{FLOOR}(A / P) * P$ .20      **Examples.** MODULO (8, 5) has the value 3. MODULO (-8, 5) has the value 2. MOD-  
21      ULO (8, -5) has the value -2. MODULO (-8, -5) has the value -3.22   **13.7.124   MOVE\_ALLOC (FROM, TO)**23      **Description.** Moves an allocation from one allocatable object to another.24      **Class.** Pure subroutine.25      **Arguments.**26      FROM               may be of any type and rank. It shall be allocatable. It is an IN-  
                          TENT(INOUT) argument.27      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.28      The allocation status of TO becomes unallocated if FROM is unallocated on entry to MOVE\_-  
29      ALLOC. Otherwise, TO becomes allocated with dynamic type, type parameters, array bounds,  
30      and value identical to those that FROM had on entry to MOVE\_ALLOC.

31      If TO has the TARGET attribute, any pointer associated with FROM on entry to MOVE\_-

ALLOC becomes correspondingly associated with TO. If TO does not have the TARGET attribute, the pointer association status of any pointer associated with FROM on entry becomes undefined.

The allocation status of FROM becomes unallocated.

#### Example.

```

REAL,ALLOCATABLE :: GRID(:),TEMPGRID(:)
...
ALLOCATE(GRID(-N:N))          ! initial allocation of GRID
...
! "reallocation" of GRID to allow intermediate points
ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
TEMPGRID(::2)=GRID ! distribute values to new locations
CALL MOVE_ALLOC(TO=GRID,FROM=TEMPGRID)
                        ! old grid is deallocated because TO is
                        ! INTENT(OUT), and GRID then "takes over"
                        ! new grid allocation

```

#### NOTE 13.18

It is expected that the implementation of allocatable objects will typically involve descriptors to locate the allocated storage; MOVE\_ALLOC could then be implemented by transferring the contents of the descriptor for FROM to the descriptor for TO and clearing the descriptor for FROM.

### 13.7.125 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

**Description.** Copies a sequence of bits from one data object to another.

**Class.** Elemental subroutine.

#### Arguments.

FROM	shall be of type integer or bits. It is an INTENT (IN) argument.
FROMPOS	shall be of type integer and nonnegative. It is an INTENT (IN) argument. FROMPOS + LEN shall be less than or equal to BIT_SIZE (FROM). The model for the interpretation of an integer value as a sequence of bits is in 13.3.
LEN	shall be of type integer and nonnegative. It is an INTENT (IN) argument.
TO	shall be a variable of the same type and kind type parameter value as FROM and may be associated with FROM (12.8.3). It is an INTENT (INOUT) argument. TO is defined by copying the sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry. The model for the interpretation of an integer value as a sequence of bits is in 13.3.
TOPOS	shall be of type integer and nonnegative. It is an INTENT (IN) argument. TOPOS + LEN shall be less than or equal to BIT_SIZE (TO).

1       **Examples.** If TO has the initial value 6, the value of TO after the statement  
 2       CALL MVBITS (7, 2, 2, TO, 0) is 5. If TO has the initial value B'000000111111', the value of  
 3       TO after the statement CALL MVBITS (B'000000000011', 0, 2, TO, 8) is B'001100111111'.

#### 4   **13.7.126   NEAREST (X, S)**

5       **Description.** Returns the nearest different machine-representable number in a given direction.

6       **Class.** Elemental function.

7       **Arguments.**

8       X                   shall be of type real.

9       S                   shall be of type real and not equal to zero.

10      **Result Characteristics.** Same as X.

11      **Result Value.** The result has a value equal to the machine-representable number distinct from  
 12      X and nearest to it in the direction of the infinity with the same sign as S.

13      **Example.** NEAREST (3.0, 2.0) has the value  $3 + 2^{-22}$  on a machine whose representation is  
 14      that of the model in Note 13.5.

##### **NOTE 13.19**

Unlike other floating-point manipulation functions, NEAREST operates on machine-representable numbers rather than model numbers. On many systems there are machine-representable numbers that lie between adjacent model numbers.

#### 15   **13.7.127   NEW\_LINE (A)**

16      **Description.** Returns a newline character.

17      **Class.** Inquiry function.

18      **Argument.** A shall be of type character. It may be a scalar or an array.

19      **Result Characteristics.** Character scalar of length one with the same kind type parameter  
 20      as A.

21      **Result Value.**

22      *Case (i):*       If A is of the default character type and the character in position 10 of the ASCII  
 23                       collating sequence is representable in the default character set, then the result is  
 24                       ACHAR(10).

25      *Case (ii):*     If A is of the ASCII character type or the ISO 10646 character type, then the  
 26                       result is CHAR(10,KIND(A)).

27      *Case (iii):*    Otherwise, the result is a processor-dependent character that represents a new-  
 28                       line in output to files connected for formatted stream output if there is such a  
 29                       character.

30      *Case (iv):*     Otherwise, the result is the blank character.

#### 31   **13.7.128   NINT (A [, KIND])**

32      **Description.** Nearest integer.

1        **Class.** Elemental function.

2        **Arguments.**

3        A                    shall be of type real.

4        KIND (optional) shall be a scalar integer initialization expression.

5        **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
6        by the value of KIND; otherwise, the kind type parameter is that of default integer type.

7        **Result Value.** The result is the integer nearest A, or if there are two integers equally near A,  
8        the result is whichever such integer has the greater magnitude.

9        **Example.** NINT (2.783) has the value 3.

## 10    13.7.129   NOT (I)

11       **Description.** Performs a bitwise complement.

12       **Class.** Elemental function.

13       **Argument.** I shall be of type integer or bits.

14       **Result Characteristics.** Same as I.

15       **Result Value.** The result has the value obtained by complementing I bit-by-bit according to  
16       the following truth table:

I	NOT (I)
1	0
0	1

17       The model for the interpretation of an integer value as a sequence of bits is in 13.3.

18       **Examples.** If I is represented by the string of bits 01010101, NOT (I) has the binary value  
19       10101010. NOT (Z'FFFF0000') has the value Z'0000FFFF'.

## 20    13.7.130   NORM2 (X)

21       **Description.**  $L_2$  norm of an array.

22       **Class.** Transformational function.

23       **Argument.** X shall be of type real. It shall be an array.

24       **Result Characteristics.** Scalar of the same type and kind type parameter value as X.

25       **Result Value.** The result has a value equal to a processor-dependent approximation to the  
26       generalized  $L_2$  norm of X, which is the square root of the sum of the squares of the elements of  
27       X.

28       **Example.** The value of NORM2 ((/ 3.0, 4.0 /)) is 5.0 (approximately).

### NOTE 13.20

It is recommended that the processor compute NORM2 in such a way that intermediate results do not overflow or underflow unless the final result would overflow or underflow, respectively.

### 1 13.7.131 NULL ([MOLD])

2 **Description.** Returns a disassociated pointer or designates an unallocated allocatable compo-  
3 nent of a structure constructor.

4 **Class.** Transformational function.

5 **Argument.** MOLD shall be a pointer or allocatable. It may be of any type or may be a  
6 procedure pointer. If MOLD is a pointer its pointer association status may be undefined,  
7 disassociated, or associated. If MOLD is allocatable its allocation status may be allocated or  
8 unallocated. It need not be defined with a value.

9 **Result Characteristics.** If MOLD is present, the characteristics are the same as MOLD. If  
10 MOLD has deferred type parameters, those type parameters of the result are deferred.

11 If MOLD is absent, the characteristics of the result are determined by the entity with which the  
12 reference is associated. See Table 13.1. MOLD shall not be absent in any other context. If any  
13 type parameters of the contextual entity are deferred, those type parameters of the result are  
14 deferred. If any type parameters of the contextual entity are assumed, MOLD shall be present.

15 If the context of the reference to NULL is an actual argument to a generic procedure, MOLD  
16 shall be present if the type, type parameters, or rank is required to resolve the generic reference.  
17 MOLD shall also be present if the reference appears as an actual argument corresponding to a  
18 dummy argument with 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

19 **Result.** The result is a disassociated pointer or an unallocated allocatable entity.

#### 20 **Examples.**

21 *Case (i):* REAL, POINTER, DIMENSION(:) :: VEC => NULL ( ) defines the initial  
22 association status of VEC to be disassociated.

23 *Case (ii):* The MOLD argument is required in the following:

```

24      INTERFACE GEN
25          SUBROUTINE S1 (J, PI)
26              INTEGER J
27              INTEGER, POINTER :: PI
28          END SUBROUTINE S1
29          SUBROUTINE S2 (K, PR)
30              INTEGER K
31              REAL, POINTER :: PR
32          END SUBROUTINE S2
33      END INTERFACE

```

```

1          REAL, POINTER :: REAL_PTR
2          CALL GEN (7, NULL (REAL_PTR) )      ! Invokes S2

```

### 3 13.7.132 NUM\_IMAGES ()

4     **Description.** Returns the number of images.

5     **Class.** Inquiry function.

6     **Argument.** None.

7     **Result Characteristics.** Default integer scalar.

8     **Result Value.** The number of images.

9     **Example.** The following code uses image 1 to read data and broadcast it to other images.

```

10  REAL :: P[*]
11  ...
12  IF (THIS_IMAGE()==1) THEN
13      READ (6,*) P
14      DO I = 2, NUM_IMAGES()
15          P[I] = P
16      END DO
17  END IF
18  SYNC_ALL

```

#### NOTE 13.21

The following two functions might be useful in connection with NUM\_IMAGES.

```

PURE INTEGER FUNCTION LOG2_IMAGES()
    ! Returns the base-2 logarithm of the number of images,
    ! truncated to an integer.
    LOG2_IMAGES = LOG(NUM_IMAGES()+0.5)/LOG(2.0)
END FUNCTION LOG2_IMAGES

PURE INTEGER FUNCTION REM_IMAGES()
    ! Returns MOD(NUM_IMAGES(),2**LOG2_IMAGES()).
    REM_IMAGES = MOD(NUM_IMAGES(),2**LOG2_IMAGES())
END FUNCTION REM_IMAGES

```

#### J3 internal note

##### Unresolved Technical Issue 69

These two extra examples are extra examples, but that don't show anything terribly useful. In particular, the "comment" in the second example is atrocious and unacceptable. Perhaps these two functions should be moved into Annex C together with a paragraph or so of witter explaining just how one might use such functions and what for (and, maybe even some code \*showing\* how they might profitably be used). In which case, leaving a note here saying that the user might find these functions useful with a forward ref would certainly be in order.



### 1 13.7.133 PACK (ARRAY, MASK [, VECTOR])

2 **Description.** Pack an array into an array of rank one under the control of a mask.

3 **Class.** Transformational function.

4 **Arguments.**

5 ARRAY may be of any type. It shall be an array.

6 MASK shall be of type logical and shall be conformable with ARRAY.

7 VECTOR shall be of the same type and type parameters as ARRAY and shall have rank one. VECTOR shall have at least as many elements as there are true elements in MASK. If MASK is scalar with the value true, VECTOR shall have at least as many elements as there are in ARRAY.

8 **Result Characteristics.** The result is an array of rank one with the same type and type parameters as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number  $t$  of true elements in MASK unless MASK is scalar with the value true, in which case the result size is the size of ARRAY.

9 **Result Value.** Element  $i$  of the result is the element of ARRAY that corresponds to the  $i$ th true element of MASK, taking elements in array element order, for  $i = 1, 2, \dots, t$ . If VECTOR is present and has size  $n > t$ , element  $i$  of the result has the value VECTOR ( $i$ ), for  $i = t + 1, \dots, n$ .

10 **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”  
 11 by the function PACK. The result of PACK (M, MASK = M /= 0) is [9, 7] and the result of  
 12 PACK (M, M /= 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is [9, 7, 6, 8, 10, 12].

### 19 13.7.134 PARITY (MASK [, DIM])

20 **Description.** Determine whether an odd number of values are true in MASK along dimension  
 21 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 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.

27 **Result Value.**

28 *Case (i):* The result of ALL (MASK) has the value true if an odd number of the elements of MASK are true, and false otherwise.

29 *Case (ii):* If MASK has rank one, PARITY (MASK, DIM) is equal to PARITY (MASK).

1                   Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of PAR-  
 2                   ITY (MASK, DIM) is equal to PARITY (MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1},$   
 3                    $\dots, s_n)$ ).

#### 4       **Examples.**

5       *Case (i):*     The value of PARITY ([T, T, T, F]) is true if T has the value true and F has the  
 6                   value false.

7       *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  
 8                   false, then PARITY (B, DIM=1) has the value [F, F, T] and PARITY (B, DIM=2)  
 9                   has the value [F, T].

### 10   **13.7.135 POPCNT (I [, KIND])**

11       **Description.** Return the number of one bits.

12       **Class.** Elemental function.

13       **Arguments.**

14       I                   shall be of type integer or bits.

15       KIND (optional) shall be a scalar integer initialization expression.

16       **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
 17       by the value of KIND; otherwise, the kind type parameter is that of the default integer type.

18       **Result Value.** If I is of type integer, the result value is equal to the number of one bits in the  
 19       sequence of bits of I. The model for the interpretation of an integer value as a sequence of bits  
 20       is in 13.3. If I is of type bits, the result value is the number of one bits in I.

21       **Examples.** POPCNT ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 2, 1, 2, 2]. POPCNT (Z'FFFF0000')  
 22       has the value 16. If B is of type bits, POPCNT (HUGE (B)) has the same value as KIND (B).

### 23   **13.7.136 POPPAR (I [, KIND])**

24       **Description.** Return the bitwise parity.

25       **Class.** Elemental function.

26       **Arguments.**

27       I                   shall be of type integer 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  
 30       by the value of KIND; otherwise, the kind type parameter is that of the default integer type.

31       **Result Value.** POPPAR (I) has the value 1 if POPCNT (I) is odd, and 0 if POPCNT (I) is  
 32       even.

33       **Examples.** POPPAR ([1, 2, 3, 4, 5, 6]) has the value [1, 1, 0, 1, 0, 0]. POPPAR (Z'FFFF0000')  
 34       has the value 0.

### 35   **13.7.137 PRECISION (X)**

**Description.** Returns the decimal precision of the model representing real numbers with the same kind type parameter as the argument.

**Class.** Inquiry function.

**Argument.** X shall be of type real or complex. It may be a scalar or an array.

**Result Characteristics.** Default integer scalar.

**Result Value.** The result has the value  $\text{INT}((p-1) * \text{LOG}_{10}(b)) + k$ , where  $b$  and  $p$  are as defined in 13.4 for the model representing real numbers with the same value for the kind type parameter as X, and where  $k$  is 1 if  $b$  is an integral power of 10 and 0 otherwise.

**Example.**  $\text{PRECISION}(X)$  has the value  $\text{INT}(23 * \text{LOG}_{10}(2.)) = \text{INT}(6.92...) = 6$  for real X whose model is as in Note 13.5.

### 13.7.138 PRESENT (A)

**Description.** Determine whether an optional argument is present.

**Class.** Inquiry function.

**Argument.** A shall be the name of an optional dummy argument that is accessible in the subprogram in which the PRESENT function reference appears. It may be of any type and it may be a pointer. It may be a scalar or an array. It may be a dummy procedure. The dummy argument A has no INTENT attribute.

**Result Characteristics.** Default logical scalar.

**Result Value.** The result has the value true if A is present (12.5.1.9) and otherwise has the value false.

### 13.7.139 PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])

**Description.** Product of all the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.

**Class.** Transformational function.

**Arguments.**

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 dummy argument.

MASK (optional) shall be of type logical and shall be conformable with ARRAY.

**Result Characteristics.** The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM 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 ARRAY.

**Result Value.**

*Case (i):* The result of PRODUCT (ARRAY) has a value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has the value one if ARRAY has size zero.

*Case (ii):* The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a

processor-dependent approximation to the product of the elements of ARRAY corresponding to the true elements of MASK or has the value one if there are no true elements.

*Case (iii):* If ARRAY has rank one, PRODUCT (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of PRODUCT (ARRAY [, MASK = MASK]). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of PRODUCT (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

PRODUCT (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  [, MASK = MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  ] ).

#### Examples.

*Case (i):* The value of PRODUCT ((/ 1, 2, 3 /)) is 6.

*Case (ii):* PRODUCT (C, MASK = C > 0.0) forms the product of the positive elements of C.

*Case (iii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , PRODUCT (B, DIM = 1) is [2, 12, 30] and PRODUCT (B, DIM = 2) is [15, 48].

### 13.7.140 RADIX (X)

**Description.** Returns the base 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 integer or real. It may be a scalar or an array.

**Result Characteristics.** Default integer scalar.

**Result Value.** The result has the value  $r$  if X is of type integer and the value  $b$  if X is of type real, where  $r$  and  $b$  are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

**Example.** RADIX (X) has the value 2 for real X whose model is as in Note 13.5.

### 13.7.141 RANDOM\_NUMBER (HARVEST)

**Description.** Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range  $0 \leq x < 1$ .

#### J3 internal note

##### Unresolved Technical Issue 70

The description does not make sense for bits. No problem with “pseudorandom number” for bits (since that’s exactly what is being generated – lots of 0s and 1s, which last time I looked were numbers). However, there is definitely a problem with requiring each bit to be less than 1!

**Class.** Subroutine.

**Argument.** HARVEST shall be of type real or bits. It is an INTENT (OUT) argument. It may be a scalar or an array variable. If it is real, it is assigned pseudorandom numbers from the uniform distribution in the interval  $0 \leq x < 1$ . If it is of type bits, it is assigned pseudorandom values with each of the KIND (HARVEST) bits of each value having a probability of approximately 0.5 of being 1.

1       **Examples.**

```

2       REAL X, Y (10, 10)
3       BITS B
4       ! Initialize X with a pseudorandom number
5       CALL RANDOM_NUMBER (HARVEST = X)
6       CALL RANDOM_NUMBER (Y)
7       ! X and Y contain uniformly distributed random numbers
8       CALL RANDOM_NUMBER(B)
9       ! B contains a uniformly random collection of 0 and 1 bits.
```

10   **13.7.142 RANDOM\_SEED ([SIZE, PUT, GET])**

11       **Description.** Restarts or queries the pseudorandom number generator used by RANDOM-  
12       NUMBER.

13       **Class.** Subroutine.

14       **Arguments.** There shall either be exactly one or no arguments present.

15       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.

16       PUT (optional) shall be a default integer array of rank one and size  $\geq N$ . It is an IN-  
TENT (IN) argument. It is used in a processor-dependent manner to compute  
the seed value accessed by the pseudorandom number generator.

17       GET (optional) shall be a default integer array of rank one and size  $\geq N$  It is an IN-  
TENT (OUT) argument. It is assigned the current value of the seed.

18       If no argument is present, the processor assigns a processor-dependent value to the seed.

19       The pseudorandom number generator used by RANDOM\_NUMBER maintains a seed that is  
20       updated during the execution of RANDOM\_NUMBER and that may be specified or returned by  
21       RANDOM\_SEED. Computation of the seed from the argument PUT is performed in a processor-  
22       dependent manner. The value returned by GET need not be the same as the value specified  
23       by PUT in an immediately preceding reference to RANDOM\_SEED. For example, following  
24       execution of the statements

```

25       CALL RANDOM_SEED (PUT=SEED1)
26       CALL RANDOM_SEED (GET=SEED2)
```

27       SEED2 need not equal SEED1. When the values differ, the use of either value as the PUT  
28       argument in a subsequent call to RANDOM\_SEED shall result in the same sequence of pseudo-  
29       random numbers being generated. For example, after execution of the statements

```

30       CALL RANDOM_SEED (PUT=SEED1)
31       CALL RANDOM_SEED (GET=SEED2)
32       CALL RANDOM_NUMBER (X1)
33       CALL RANDOM_SEED (PUT=SEED2)
34       CALL RANDOM_NUMBER (X2)
```

1 X2 equals X1.

2 **Examples.**

```

3      CALL RANDOM_SEED                      ! Processor initialization
4      CALL RANDOM_SEED (SIZE = K)           ! Puts size of seed in K
5      CALL RANDOM_SEED (PUT = SEED (1 : K)) ! Define seed
6      CALL RANDOM_SEED (GET = OLD (1 : K))  ! Read current seed

```

### 7 13.7.143 RANGE (X)

8 **Description.** Returns the decimal exponent range of the model representing integer or real  
9 numbers with the same kind type parameter as the argument.

10 **Class.** Inquiry function.

11 **Argument.** X shall be of type integer, real, or complex. It may be a scalar or an array.

12 **Result Characteristics.** Default integer scalar.

13 **Result Value.**

14 *Case (i):* For an integer argument, the result has the value INT (LOG10 (HUGE(X))).

15 *Case (ii):* For a real argument, the result has the value INT (MIN (LOG10 (HUGE(X)),  
16 -LOG10 (TINY(X)))).

17 *Case (iii):* For a complex argument, the result has the value RANGE(REAL(X)).

18 **Examples.** RANGE (X) has the value 38 for real X whose model is as in Note 13.5, because  
19 in this case  $\text{HUGE}(X) = (1 - 2^{-24}) \times 2^{127}$  and  $\text{TINY}(X) = 2^{-127}$ .

### 20 13.7.144 REAL (A [, KIND])

21 **Description.** Convert to real type.

22 **Class.** Elemental function.

23 **Arguments.**

24 A shall be of type integer, real, complex, or bits.

25 KIND (optional) shall be a scalar integer initialization expression.

26 **Result Characteristics.** Real.

27 *Case (i):* If A is of type integer, real, or bits, and KIND is present, the kind type parameter  
28 is that specified by the value of KIND. If A is of type integer, real, or bits, and  
29 KIND is not present, the kind type parameter is that of default real type.

30 *Case (ii):* If A is of type complex and KIND is present, the kind type parameter is that  
31 specified by the value of KIND. If A is of type complex and KIND is not present,  
32 the kind type parameter is the kind type parameter of A.

33 **Result Value.**

34 *Case (i):* If A is of type integer or real, the result is equal to a processor-dependent ap-  
35 proximation to A.

- 1        *Case (ii):*    If A is of type complex, the result is equal to a processor-dependent approximation  
 2                    to the real part of A.
- 3        *Case (iii):*    If A is of type bits and KIND (A) is greater than or equal to BITS\_KIND (*result*),  
 4                    the result has the value whose physical representation is the same as the rightmost  
 5                    bits of A.
- 6        *Case (iv):*    If A is of type bits and KIND (A) is less than BITS\_KIND (*result*), the rightmost  
 7                    KIND (A) bits of the physical representation of the result value are the same as  
 8                    those of A, and the remaining bits of the physical representation of the result value  
 9                    are zero.

10        **Examples.** REAL (-3) has the value -3.0. REAL (Z) has the same kind type parameter and  
 11        the same value as the real part of the complex variable Z. REAL (Z'7F80000') has the value  
 12        positive infinity if the default real kind is IEEE single precision.

### 13    13.7.145    REPEAT (STRING, NCOPIES)

14        **Description.** Concatenate several copies of a string.

15        **Class.** Transformational function.

16        **Arguments.**

17        STRING            shall be scalar and of type character.

18        NCOPIES           shall be scalar and of type integer. Its value shall not be negative.

19        **Result Characteristics.** Character scalar of length NCOPIES times that of STRING, with  
 20        the same kind type parameter as STRING.

21        **Result Value.** The value of the result is the concatenation of NCOPIES copies of STRING.

22        **Examples.** REPEAT ('H', 2) has the value HH. REPEAT ('XYZ', 0) has the value of a zero-  
 23        length string.

### 24    13.7.146    RESHAPE (SOURCE, SHAPE [, PAD, ORDER])

25        **Description.** Constructs an array of a specified shape from the elements of a given array.

26        **Class.** Transformational function.

27        **Arguments.**

28        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.

29        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.

30        PAD (optional)    shall be of the same type and type parameters as SOURCE. It shall be an  
                              array.

31        ORDER            shall be of type integer, shall have the same shape as SHAPE, and its value  
                              shall be a permutation of (1, 2, ..., *n*), where *n* is the size of SHAPE. If  
                              absent, it is as if it were present with value (1, 2, ..., *n*).

32        **Result Characteristics.** The result is an array of shape SHAPE (that is, SHAPE (RE-  
 33        SHAPE (SOURCE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type

parameters as SOURCE.

**Result Value.** The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER ( $n$ ), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

**Examples.** RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /)) has the value  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ . RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 4 /), (/ 0, 0 /), (/ 2, 1 /)) has the value  $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$ .

### 13.7.147 RRSPACING (X)

**Description.** Returns the reciprocal of the relative spacing of model numbers near the argument value.

**Class.** Elemental function.

**Argument.** X shall be of type real.

**Result Characteristics.** Same as X.

**Result Value.** The result has the value  $|X \times b^{-e}| \times b^p$ , where  $b$ ,  $e$ , and  $p$  are as defined in 13.4 for the model representation of X. If X is an IEEE infinity, the result is zero. If X is an IEEE NaN, the result is that NaN.

**Example.** RRSPACING (−3.0) has the value  $0.75 \times 2^{24}$  for reals whose model is as in Note 13.5.

### 13.7.148 SAME\_TYPE\_AS (A, B)

**Description.** Inquires whether the dynamic type of A is the same as the dynamic type of B.

**Class.** Inquiry function.

**Arguments.**

A shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

B shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

**Result Characteristics.** Default logical scalar.

**Result Value.** The result is true if and only if the dynamic type of A is the same as the dynamic type of B.

#### NOTE 13.22

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

### 13.7.149 SCALE (X, I)

**Description.** Returns  $X \times b^I$  where  $b$  is the base of the model representation of X.

**Class.** Elemental function.



1       **Arguments.**

2       X                   shall be of type real.

3       I                   shall be of type integer.

4       **Result Characteristics.** Same as X.5       **Result Value.** The result has the value  $X \times b^I$ , where  $b$  is defined in 13.4 for model numbers  
6       representing values of X, provided this result is within range; if not, the result is processor  
7       dependent.8       **Example.** SCALE (3.0, 2) has the value 12.0 for reals whose model is as in Note 13.5.9   **13.7.150   SCAN (STRING, SET [, BACK, KIND])**10       **Description.** Scan a string for any one of the characters in a set of characters.11       **Class.** Elemental function.12       **Arguments.**

13       STRING            shall be of type character.

14       SET               shall be of type character with the same kind type parameter as STRING.

15       BACK (optional) shall be of type logical.

16       KIND (optional) shall be a scalar integer initialization expression.

17       **Result Characteristics.** Integer. 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.**20       *Case (i):*       If BACK is absent or is present with the value false and if STRING contains at  
21       least one character that is in SET, the value of the result is the position of the  
22       leftmost character of STRING that is in SET.23       *Case (ii):*      If BACK is present with the value true and if STRING contains at least one  
24       character that is in SET, the value of the result is the position of the rightmost  
25       character of STRING that is in SET.26       *Case (iii):*     The value of the result is zero if no character of STRING is in SET or if the  
27       length of STRING or SET is zero.28       **Examples.**29       *Case (i):*       SCAN ('FORTRAN', 'TR') has the value 3.30       *Case (ii):*      SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.31       *Case (iii):*     SCAN ('FORTRAN', 'BCD') has the value 0.32   **13.7.151   SELECTED\_BITS\_KIND (N)**33       **Description.** Returns a value of the kind type parameter of a bits type with N bits.34       **Class.** Transformational function.35       **Argument.** N shall be scalar and of type integer.

1       **Result Characteristics.** Default integer scalar.

2       **Result Value.** The result value is  $N$  if the processor supports a bits type with a kind type  
3 parameter equal to  $N$ ; otherwise the result value is  $-1$ .

4       **Example.** If the value of `NUMERIC_STORAGE_SIZE` is 32, `SELECT_BITS_KIND` (43) has  
5 the value 43.

## 6   13.7.152   **SELECTED\_CHAR\_KIND (NAME)**

7       **Description.** Returns the value of the kind type parameter of the character set named by the  
8 argument.

9       **Class.** Transformational function.

10      **Argument.** `NAME` shall be scalar and of type default character.

11      **Result Characteristics.** Default integer scalar.

12      **Result Value.** If `NAME` has the value `DEFAULT`, then the result has a value equal to that of  
13 the kind type parameter of the default character type. If `NAME` has the value `ASCII`, then the  
14 result has a value equal to that of the kind type parameter of the ASCII character type if the  
15 processor supports such a type; otherwise the result has the value  $-1$ . If `NAME` has the value  
16 `ISO_10646`, then the result has a value equal to that of the kind type parameter of the ISO/IEC  
17 10646-1:2000 UCS-4 character type if the processor supports such a type; otherwise the result  
18 has the value  $-1$ . If `NAME` is a processor-defined name of some other character type supported  
19 by the processor, then the result has a value equal to that of the kind type parameter of that  
20 character type. If `NAME` is not the name of a supported character type, then the result has the  
21 value  $-1$ . The `NAME` is interpreted without respect to case or trailing blanks.

22      **Examples.** `SELECTED_CHAR_KIND ('ASCII')` has the value 1 on a processor that uses 1  
23 as the kind type parameter for the ASCII character set. The following subroutine produces a  
24 Japanese date stamp.

```
25       SUBROUTINE create_date_string(string)
26           INTRINSIC date_and_time,selected_char_kind
27           INTEGER,PARAMETER :: ucs4 = selected_char_kind("ISO_10646")
28           CHARACTER(1,UCS4),PARAMETER :: nen=CHAR(INT(Z'5e74'),UCS4), & !year
29               gatsu=CHAR(INT(Z'6708'),UCS4), & !month
30               nichi=CHAR(INT(Z'65e5'),UCS4) !day
31           CHARACTER(len= *, kind= ucs4) string
32           INTEGER values(8)
33           CALL date_and_time(values=values)
34           WRITE(string,1) values(1),nen,values(2),gatsu,values(3),nichi
35       1 FORMAT(IO,A,IO,A,IO,A)
36       END SUBROUTINE"
```

## 37   13.7.153   **SELECTED\_INT\_KIND (R)**

38      **Description.** Returns a value of the kind type parameter of an integer type that represents all  
39 integer values  $n$  with  $-10^R < n < 10^R$ .

40      **Class.** Transformational function.

1       **Argument.** R shall be scalar and of type integer.

2       **Result Characteristics.** Default integer scalar.

3       **Result Value.** The result has a value equal to the value of the kind type parameter of an  
 4 integer type that represents all values  $n$  in the range  $-10^R < n < 10^R$ , or if no such kind type  
 5 parameter is available on the processor, the result is  $-1$ . If more than one kind type parameter  
 6 meets the criterion, the value returned is the one with the smallest decimal exponent range,  
 7 unless there are several such values, in which case the smallest of these kind values is returned.

8       **Example.** Assume a processor supports two integer kinds, 32 with representation method  
 9  $r = 2$  and  $q = 31$ , and 64 with representation method  $r = 2$  and  $q = 63$ . On this processor  
 10 `SELECTED_INT_KIND(9)` has the value 32 and `SELECTED_INT_KIND(10)` has the value 64.

### 11 13.7.154 `SELECTED_REAL_KIND ([P, R, RADIX])`

12       **Description.** Returns a value of the kind type parameter of a real type with decimal precision  
 13 of at least P digits, a decimal exponent range of at least R, and a radix of RADIX.

14       **Class.** Transformational function.

15       **Arguments.** At least one argument shall be present.

16       P (optional)       shall be scalar and of type integer.

17       R (optional)       shall be scalar and of type integer.

18       RADIX (optional) shall be scalar and of type integer.

19       **Result Characteristics.** Default integer scalar.

20       **Result Value.** If P or R is absent, the result value is the same as if it were present with the  
 21 value zero. If RADIX is absent, there is no requirement on the radix of the selected kind.

22 The result has a value equal to a value of the kind type parameter of a real type with decimal precision,  
 23 as returned by the function `PRECISION`, of at least P digits, a decimal exponent range, as returned by  
 24 the function `RANGE`, of at least R, and a radix, as returned by the function `RADIX`, of RADIX, if such  
 25 a kind type parameter is available on the processor.

26 Otherwise, the result is  $-1$  if the processor supports a real type with radix RADIX and exponent range  
 27 of at least R but not with precision of at least P,  $-2$  if the processor supports a real type with radix  
 28 RADIX and precision of at least P but not with exponent range of at least R,  $-3$  if the processor  
 29 supports a real type with radix RADIX but with neither precision of at least P nor exponent range of  
 30 at least R,  $-4$  if the processor supports a real type with radix RADIX and either precision of at least  
 31 P or exponent range of at least R but not both together, and  $-5$  if the processor supports no real type  
 32 with radix RADIX.

33 If more than one kind type parameter value meets the criteria, the value returned is the one with the  
 34 smallest decimal precision, unless there are several such values, in which case the smallest of these kind  
 35 values is returned.

36       **Example.** `SELECTED_REAL_KIND (6, 70)` has the value `KIND (0.0)` on a machine that  
 37 supports a default real approximation method with  $b = 16$ ,  $p = 6$ ,  $e_{\min} = -64$ , and  $e_{\max} = 63$   
 38 and does not have a less precise approximation method.

### 39 13.7.155 `SET_EXPONENT (X, I)`

40       **Description.** Returns the model number whose fractional part is the fractional part of the

1 model representation of  $X$  and whose exponent part is  $I$ .

2 **Class.** Elemental function.

3 **Arguments.**

4  $X$  shall be of type real.

5  $I$  shall be of type integer.

6 **Result Characteristics.** Same as  $X$ .

7 **Result Value.** The result has the value  $X \times b^{I-e}$ , where  $b$  and  $e$  are as defined in 13.4 for the  
8 model representation of  $X$ . If  $X$  has value zero, the result has value zero.

9 **Example.** SET\_EXPONENT (3.0, 1) has the value 1.5 for reals whose model is as in Note  
10 13.5.

### 11 13.7.156 SHAPE (SOURCE [, KIND])

12 **Description.** Returns the shape of an array or a scalar.

13 **Class.** Inquiry function.

14 **Arguments.**

15 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.

16 KIND (optional) shall be a scalar integer initialization expression.

17 **Result Characteristics.** Integer. 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. The  
19 result is an array of rank one whose size is equal to the rank of SOURCE.

20 **Result Value.** The value of the result is the shape of SOURCE.

21 **Examples.** The value of SHAPE (A (2:5, -1:1) ) is [4, 3]. The value of SHAPE (3) is the  
22 rank-one array of size zero.

### 23 13.7.157 SHIFTA (I, SHIFT)

24 **Description.** Performs an arithmetic right shift.

25 **Class.** Elemental function.

26 **Arguments.**

27  $I$  shall be of type integer or bits.

28 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to  
BIT\_SIZE (I).

29 **Result Characteristics.** Same as  $I$ .

30 **Result Value.** The result has the value obtained by shifting the bits of  $I$  to the right SHIFT  
31 bits and replicating the leftmost bit of  $I$  in the left SHIFT bits.

**J3 internal note**

## Unresolved Technical Issue 71

This does not just assume integers are binary, but that they are twos complement binary. Other machines have existed and I believe even still exist. If this is meant to be an *arithmetic* shift surely it should give the right answer on all machines? After all, if we are only going to support binary twos complement, we can define negative integer bit patterns (and we have not done that, nor should we).

NOTE: I am complaining about the **Description**, not about the functionality. Or maybe I'm complaining that the functionality doesn't match my signed-magnitude machines version. Who knows.

- 1 If SHIFT is zero the result is I. Bits shifted out from the right are lost. The model for the interpretation
- 2 of an integer value as a sequence of bits is in 13.3.

**J3 internal note**

## Unresolved Technical Issue 72

That model does not interpret values with the top bit set, i.e. is completely irrelevant to this function. 13.3 says that if it has the top bit set, the interpretation is "processor dependent"; this is not useful for this function. Maybe there is no fix possible here...

- 3 **Example.** SHIFTA (B'10000', 2) has the value B'11100'.

#### 4 **13.7.158 SHIFTL (I, SHIFT)**

- 5 **Description.** Performs a left shift.

- 6 **Class.** Elemental function.

- 7 **Arguments.**

- 8 I shall be of type integer or bits.

- 9 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT\_SIZE (I).

- 10 **Result Characteristics.** Same as I.

- 11 **Result Value.** The result has a value equal to ISHFT (I, SHIFT).

- 12 **Examples.** SHIFTL (3, 1) has the value 6. SHIFTL (B'00001', 2) has the value B'00100'.

#### 13 **13.7.159 SHIFTR (I, SHIFT)**

- 14 **Description.** Performs a right shift.

- 15 **Class.** Elemental function.

- 16 **Arguments.**

- 17 I shall be of type integer or bits.

- 18 SHIFT shall be of type integer. It shall be nonnegative and less than or equal to BIT\_SIZE (I).

- 19 **Result Characteristics.** Same as I.

- 20 **Result Value.** The value of the result is ISHFT (I, -SHIFT).

1       **Examples.** SHIFTR (3, 1) has the value 1. SHIFTR (B'10000', 2) has the value B'00100'.

## 2   **13.7.160   SIGN (A, B)**

3       **Description.** Magnitude of A with the sign of B.

4       **Class.** Elemental function.

5       **Arguments.**

6       A                   shall be of type integer or real.

7       B                   shall be of the same type and kind type parameter as A.

8       **Result Characteristics.** Same as A.

9       **Result Value.**

10      *Case (i):*       If  $B > 0$ , the value of the result is  $|A|$ .

11      *Case (ii):*      If  $B < 0$ , the value of the result is  $-|A|$ .

12      *Case (iii):*     If B is of type integer and  $B=0$ , the value of the result is  $|A|$ .

13      *Case (iv):*     If B is of type real and is zero, then

14                   (1)   If the processor cannot distinguish between positive and negative real zero,  
15                       the value of the result is  $|A|$ .

16                   (2)   If B is positive real zero, the value of the result is  $|A|$ .

17                   (3)   If B is negative real zero, the value of the result is  $-|A|$ .

18      **Example.** SIGN (-3.0, 2.0) has the value 3.0.

## 19   **13.7.161   SIN (X)**

20      **Description.** Sine function.

21      **Class.** Elemental function.

22      **Argument.** X shall be of type real or complex.

23      **Result Characteristics.** Same as X.

24      **Result Value.** The result has a value equal to a processor-dependent approximation to  $\sin(X)$ .  
25      If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is  
26      regarded as a value in radians.

27      **Example.** SIN (1.0) has the value 0.84147098 (approximately).

## 28   **13.7.162   SINH (X)**

29      **Description.** Hyperbolic sine function.

30      **Class.** Elemental function.

31      **Argument.** X shall be of type real or complex.

32      **Result Characteristics.** Same as X.

33      **Result Value.** The result has a value equal to a processor-dependent approximation to  $\sinh(X)$ .  
34      If X is of type complex its imaginary part is regarded as a value in radians.

1       **Example.** SINH (1.0) has the value 1.1752012 (approximately).

## 2   **13.7.163   SIZE (ARRAY [, DIM, KIND])**

3       **Description.** Returns the extent of an array along a specified dimension or the total number  
4       of elements in the array.

5       **Class.** Inquiry function.

6       **Arguments.**

7       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.

8       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.

9       KIND (optional) shall be a scalar integer initialization expression.

10      **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that  
11      specified by the value of KIND; otherwise the kind type parameter is that of default integer  
12      type.

13      **Result Value.** The result has a value equal to the extent of dimension DIM of ARRAY or, if  
14      DIM is absent, the total number of elements of ARRAY.

15      **Examples.** The value of SIZE (A (2:5, -1:1), DIM=2) is 3. The value of SIZE (A (2:5, -1:1))  
16      is 12.

## 17   **13.7.164   SPACING (X)**

18      **Description.** Returns the absolute spacing of model numbers near the argument value.

19      **Class.** Elemental function.

20      **Argument.** X shall be of type real.

21      **Result Characteristics.** Same as X.

22      **Result Value.** If X does not have the value zero, the result has the value  $b^{\max(e-p, e_{\text{MIN}}-1)}$ ,  
23      where  $b$ ,  $e$ , and  $p$  are as defined in 13.4 for the model representation of X. If X has the value  
24      zero, the result is the same as that of TINY (X). If X is an IEEE infinity, the result is positive  
25      infinity. If X is an IEEE NaN, the result is that NaN.

26      **Example.** SPACING (3.0) has the value  $2^{-22}$  for reals whose model is as in Note 13.5.

## 27   **13.7.165   SPREAD (SOURCE, DIM, NCOPIES)**

28      **Description.** Replicates an array by adding a dimension. Broadcasts several copies of SOURCE  
29      along a specified dimension (as in forming a book from copies of a single page) and thus forms  
30      an array of rank one greater.

31      **Class.** Transformational function.

32      **Arguments.**

33      SOURCE           may be of any type. It may be a scalar or an array. The rank of SOURCE  
                          shall be less than 15.

1 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.

2 NCOPIES shall be scalar and of type integer.

3 **Result Characteristics.** The result is an array of the same type and type parameters as  
4 SOURCE and of rank  $n + 1$ , where  $n$  is the rank of SOURCE.

5 *Case (i):* If SOURCE is scalar, the shape of the result is  $(\text{MAX}(\text{NCOPIES}, 0))$ .

6 *Case (ii):* If SOURCE is an array with shape  $(d_1, d_2, \dots, d_n)$ , the shape of the result is  
7  $(d_1, d_2, \dots, d_{\text{DIM}-1}, \text{MAX}(\text{NCOPIES}, 0), d_{\text{DIM}}, \dots, d_n)$ .

8 **Result Value.**

9 *Case (i):* If SOURCE is scalar, each element of the result has a value equal to SOURCE.

10 *Case (ii):* If SOURCE is an array, the element of the result with subscripts  $(r_1, r_2, \dots, r_{n+1})$   
11 has the value SOURCE  $(r_1, r_2, \dots, r_{\text{DIM}-1}, r_{\text{DIM}+1}, \dots, r_{n+1})$ .

12 **Examples.** If A is the array [2, 3, 4], SPREAD (A, DIM=1, NCOPIES=NC) is the array

13  $\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$  if NC has the value 3 and is a zero-sized array if NC has the value 0.

## 14 13.7.166 SQRT (X)

15 **Description.** Square root.

16 **Class.** Elemental function.

17 **Argument.** X shall be of type real or complex. Unless X is complex, its value shall be greater  
18 than or equal to zero.

19 **Result Characteristics.** Same as X.

20 **Result Value.** The result has a value equal to a processor-dependent approximation to the  
21 square root of X. A result of type complex is the principal value with the real part greater than  
22 or equal to zero. When the real part of the result is zero, the imaginary part has the same sign  
23 as the imaginary part of X.

24 **Example.** SQRT (4.0) has the value 2.0 (approximately).

## 25 13.7.167 SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])

26 **Description.** Sum all the elements of ARRAY along dimension DIM corresponding to the true  
27 elements of MASK.

28 **Class.** Transformational function.

29 **Arguments.**

30 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  
31 be an optional dummy argument.

32 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

33 **Result Characteristics.** The result is of the same type and kind type parameter as AR-



RAY. 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 ARRAY.

### Result Value.

*Case (i):* The result of SUM (ARRAY) has a value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has the value zero if ARRAY has size zero.

*Case (ii):* The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the true elements of MASK or has the value zero if there are no true elements.

*Case (iii):* If ARRAY has rank one, SUM (ARRAY, DIM = DIM [, MASK = MASK]) has a value equal to that of SUM (ARRAY [, MASK = MASK]). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  SUM (ARRAY, DIM = DIM [, MASK = MASK]) is equal to

$$\text{SUM}(\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) [, \text{MASK} = \text{MASK}(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) ] ).$$

### Examples.

*Case (i):* The value of SUM ((/ 1, 2, 3 /)) is 6.

*Case (ii):* SUM (C, MASK= C > 0.0) forms the sum of the positive elements of C.

*Case (iii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , SUM (B, DIM = 1) is [3, 7, 11] and SUM (B, DIM = 2) is [9, 12].

## 13.7.168 SYSTEM\_CLOCK ([COUNT, COUNT\_RATE, COUNT\_MAX])

**Description.** Returns numeric data from a real-time clock.

**Class.** Subroutine.

### Arguments.

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.

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.

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.

**Example.** If the processor clock is a 24-hour clock that registers time at approximately 18.20648193 ticks per second, at 11:30 A.M. the reference

CALL SYSTEM\_CLOCK (COUNT = C, COUNT\_RATE = R, COUNT\_MAX = M)

defines  $C = (11 \times 3600 + 30 \times 60) \times 18.20648193 = 753748$ ,  $R = 18.20648193$ , and  $M =$

1  $24 \times 3600 \times 18.20648193 - 1 = 1573039.$

## 2 **13.7.169 TAN (X)**

3 **Description.** Tangent 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  $\tan(X)$ .  
 8 If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is  
 9 regarded as a value in radians.

10 **Example.** TAN (1.0) has the value 1.5574077 (approximately).

## 11 **13.7.170 TANH (X)**

12 **Description.** Hyperbolic tangent function.

13 **Class.** Elemental function.

14 **Argument.** X shall be of type real or complex.

15 **Result Characteristics.** Same as X.

16 **Result Value.** The result has a value equal to a processor-dependent approximation to  $\tanh(X)$ .  
 17 If X is of type complex its imaginary part is regarded as a value in radians.

18 **Example.** TANH (1.0) has the value 0.76159416 (approximately).

## 19 **13.7.171 THIS\_IMAGE () or THIS\_IMAGE (CO\_ARRAY [, DIM])**

20 **Description.** Returns the index of the invoking image, a single co-subscript, or a list of co-  
 21 subscripts.

22 **Class.** Inquiry function.

23 **Arguments.**

24 CO\_ARRAY shall be a co-array and may be of any type.  
 (optional)

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  
 25 absent optional dummy argument.

26 **Result Characteristics.** Type default integer. It is scalar if CO\_ARRAY is absent or DIM is  
 27 present; otherwise, the result has rank one and its size is equal to the co-rank of CO\_ARRAY.

28 **Result Value.**

29 *Case (i):* The result of THIS\_IMAGE () is a scalar with a value equal to the index of the  
 30 invoking image.

31 *Case (ii):* The result of THIS\_IMAGE (CO\_ARRAY) is the sequence of co-subscript values  
 32 for CO\_ARRAY that would specify the invoking image.

1        *Case (iii):*    The result of THIS\_IMAGE (CO\_ARRAY, DIM) is the value of co-subscript DIM  
 2                            in the sequence of co-subscript values for CO\_ARRAY that would specify the  
 3                            invoking image.

4            **Examples.** If A is declared by the statement REAL A (10, 20) [10, 0:9, 0:\*]  
 5            then on image 5, THIS\_IMAGE () has the value 5 and THIS\_IMAGE (A) has the value [5, 0, 0].  
 6            For the same co-array on image 213, THIS\_IMAGE (A) has the value [3, 1, 2].  
 7            The following code uses image 1 to read data. The other images then copy the data.

```
8            IF (THIS_IMAGE()==1) READ (*,*) P
9            SYNC_ALL
10           P = P[1]
```

#### NOTE 13.23

For an example of a module that implements a function similar to the intrinsic IMAGE\_INDEX, see subclause C.10.1.

### 11 13.7.172 TINY (X)

12        **Description.** Returns the smallest positive number of the model representing numbers of the  
 13                            same type and kind type parameter as the argument.

14        **Class.** Inquiry function.

15        **Argument.** X shall be of type real. It may be a scalar or an array.

16        **Result Characteristics.** Scalar with the same type and kind type parameter as X.

17        **Result Value.** The result has the value  $b^{e_{\min}-1}$  where  $b$  and  $e_{\min}$  are as defined in 13.4 for the  
 18                            model representing numbers of the same type and kind type parameter as X.

19        **Example.** TINY (X) has the value  $2^{-127}$  for real X whose model is as in Note 13.5.

### 20 13.7.173 TRAILZ (I [, KIND])

21        **Description.** Count the number of trailing zero bits.

22        **Class.** Elemental function.

23        **Arguments.**

24        I                    shall be of type integer or bits.

25        KIND (optional) shall be a scalar integer initialization expression.

26        **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
 27                            by the value of KIND; otherwise, the kind type parameter is that of default integer type.

28        **Result Value.** If all of the bits of I are zero, the result value is BIT\_SIZE (I). Otherwise, the  
 29                            result value is the position of the rightmost 1 bit in I. The model for the interpretation of an  
 30                            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.

### 13.7.174 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 value 3.3. The imaginary part of the second element is processor dependent.

*Case (iii):*      TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /), 1) is a complex rank-one array of length one whose only element has the value (1.1, 2.2).

### 13.7.175 TRANSPOSE (MATRIX)

**Description.** Transpose an array of rank two.

**Class.** Transformational function.

**Argument.** MATRIX may be of any type and shall have rank two.

**Result Characteristics.** The result is an array of the same type and type parameters as MATRIX and with rank two and shape  $(n, m)$  where  $(m, n)$  is the shape of MATRIX.

**Result Value.** Element  $(i, j)$  of the result has the value MATRIX  $(j, i)$ ,  $i = 1, 2, \dots, n$ ;  $j = 1, 2, \dots, m$ .

**Example.** If A is the array  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ , then TRANSPOSE (A) has the value  $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$ .

### 13.7.176 TRIM (STRING)

**Description.** Returns the argument with trailing blank characters removed.

**Class.** Transformational function.

**Argument.** STRING shall be of type character and shall be scalar.

**Result Characteristics.** Character with the same kind type parameter value as STRING and with a length that is the length of STRING less the number of trailing blanks in STRING.

**Result Value.** The value of the result is the same as STRING except any trailing blanks are removed. If STRING contains no nonblank characters, the result has zero length.

**Example.** TRIM ( ' A B ' ) has the value ' A B '.

### 13.7.177 UBOUND (ARRAY [, DIM, KIND])

**Description.** Returns all the upper bounds of an array or a specified upper bound.

**Class.** Inquiry function.

**Arguments.**

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.

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.

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 rank of ARRAY.

**Result Value.**

*Case (i):* For an array section or for an array expression, other than a whole array or array structure component, UBOUND (ARRAY, DIM) has a value equal to the number of elements in the given dimension; otherwise, it has a value equal to the upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero.

*Case (ii):* UBOUND (ARRAY) has a value whose  $i$ th element is equal to UBOUND (ARRAY,  $i$ ), for  $i = 1, 2, \dots, n$ , where  $n$  is the rank of ARRAY.

1       **Examples.** If A is declared by the statement  
 2       REAL A (2:3, 7:10)  
 3       then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

#### 4   **13.7.178 UNPACK (VECTOR, MASK, FIELD)**

5       **Description.** Unpack an array of rank one into an array under the control of a mask.

6       **Class.** Transformational function.

7       **Arguments.**

8       VECTOR       may be of any type. It shall have rank one. Its size shall be at least  $t$  where  
                        $t$  is the number of true elements in MASK.

9       MASK         shall be an array of type logical.

10      FIELD        shall be of the same type and type parameters as VECTOR and shall be  
                       conformable with MASK.

11      **Result Characteristics.** The result is an array of the same type and type parameters as  
 12      VECTOR and the same shape as MASK.

13      **Result Value.** The element of the result that corresponds to the  $i$ th true element of MASK,  
 14      in array element order, has the value VECTOR ( $i$ ) for  $i = 1, 2, \dots, t$ , where  $t$  is the number of  
 15      true values in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to  
 16      the corresponding element of FIELD if it is an array.

17      **Examples.** Particular values may be “scattered” to particular positions in an array by us-  
 18      ing UNPACK. If M is the array  $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ , V is the array [1, 2, 3], and Q is the logical

19      mask  $\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$ , where “T” represents true and “.” represents false, then the result of

20      UNPACK (V, MASK = Q, FIELD = M) has the value  $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$  and the result of UN-

21      PACK (V, MASK = Q, FIELD = 0) has the value  $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ .

#### 22   **13.7.179 VERIFY (STRING, SET [, BACK, KIND])**

23      **Description.** Verify that a set of characters contains all the characters in a string by identifying  
 24      the position of the first character in a string of characters that does not appear in a given set of  
 25      characters.

26      **Class.** Elemental function.

27      **Arguments.**

28      STRING        shall be of type character.

29      SET           shall be of type character with the same kind type parameter as STRING.

30      BACK (optional) shall be of type logical.

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  
3       by the value of KIND; otherwise the kind type parameter is that of default integer type.

4       **Result Value.**

5       *Case (i):*     If BACK is absent or has the value false and if STRING contains at least one  
6       character that is not in SET, the value of the result is the position of the leftmost  
7       character of STRING that is not in SET.

8       *Case (ii):*    If BACK is present with the value true and if STRING contains at least one  
9       character that is not in SET, the value of the result is the position of the rightmost  
10      character of STRING that is not in SET.

11      *Case (iii):*   The value of the result is zero if each character in STRING is in SET or if STRING  
12      has zero length.

13      **Examples.**

14      *Case (i):*     VERIFY ('ABBA', 'A') has the value 2.

15      *Case (ii):*    VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

16      *Case (iii):*   VERIFY ('ABBA', 'AB') has the value 0.

## 17   13.8   Standard intrinsic modules

### 18   13.8.1   General

19   This part of ISO/IEC 1539 defines five standard intrinsic modules: a Fortran environment module, a  
20   set of three modules to support exception handling and IEEE arithmetic, and a module to support  
21   interoperability with the C programming language.

22   A processor may extend the standard intrinsic modules to provide public entities in them in addition to  
23   those specified in this standard.

#### 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.

### 24   13.8.2   The IEEE modules

25   The IEEE\_EXCEPTIONS, IEEE\_ARITHMETIC, and IEEE\_FEATURES intrinsic modules are described in Clause 14.  
26

### 27   13.8.3   The ISO\_FORTRAN\_ENV intrinsic module

#### 28   13.8.3.1   General

29   The intrinsic module ISO\_FORTRAN\_ENV provides public entities relating to the Fortran environment.

30   The processor shall provide the named constants, derived type, and procedures described in subclause  
31   13.8.3.

### 1 13.8.3.2 CHARACTER\_STORAGE\_SIZE

2 The value of the default integer scalar constant CHARACTER\_STORAGE\_SIZE is the size expressed  
3 in bits of the character storage unit (16.5.3.2).

### 4 13.8.3.3 COMPILER\_OPTIONS ()

5 **Description.** Returns a processor-dependent string describing the options that controlled the  
6 program translation phase.

7 **Class.** Inquiry function.

8 **Argument.** None.

9 **Result Characteristics.** Default character scalar with processor-dependent length.

10 **Result Value.** A processor-dependent value which describes the options that controlled the  
11 translation phase of program execution.

12 **Example.** COMPILER\_OPTIONS () might have the value  
13 '/OPTIMIZE /FLOAT=IEEE /CHECK:SUBSCRIPTS'.

### 14 13.8.3.4 COMPILER\_VERSION ()

15 **Description.** Returns a processor-dependent string identifying the program translation phase.

16 **Class.** Inquiry function.

17 **Argument.** None.

18 **Result Characteristics.** Default character scalar with processor-dependent length.

19 **Result Value.** A processor-dependent value that identifies the name and version of the program  
20 translation phase of the processor.

21 **Example.** COMPILER\_VERSION () might have the value  
22 'Super-fast KL-10 Compiler Version 0.1'.

#### NOTE 13.25

For both COMPILER\_OPTIONS and COMPILER\_VERSIONS 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 13.8.3.5 IMAGE\_TEAM

24 A scalar object of type IMAGE\_TEAM identifies a team of images. It shall have only private components.



**J3 internal note**

Unresolved Technical Issue 73

Email from ‘‘the subgroup’’ follows:

```
> *** Teams are always built locally from an image list using the
> FORM_TEAM intrinsic. You would not want to use in this image a team
> variable value from a different image. The values may differ among the
> images of a team. These variables might contain detailed information
> on such things as which images are its neighbors in the hardware, or
> the location of the image in a reduction tree.
>
> As you noted, the normative text supporting Note 13.19 is missing. This
> was removed recently, but we feel it should be put back. The "special"
> derived types IMAGE_TEAM, C_PTR, and C_FUNPTR are very likely to contain
> information that is specific to the image where it was defined. Use of
> these data on a different image could lead to various errors. The
> changes required are:
>
> 1) Restore Note 13.19.
>
> 2) Modify the edit for [50:25+] to add another constraint:
>
> "C440d (R440) A component of derived type image_team (13.8.2.x), c_ptr,
> or c_funptr (15.2.2) shall not be a co-array."
>
> 3) In the edit at [73:1+] replace the text for C517b with:
>
> "C517b (R501) A co-array shall not be of derived type image_team
> (13.8.2.x) or c_ptr (15.2.2) and shall not have the POINTER attribute."
>
> 4) In the edit at [105:9+] replace the text for C613b with:
>
> "C613b (R613) A <data-ref> that is a co-indexed object shall not be of a
> type that has a pointer ultimate component or has a subcomponent of
> derived type image_team (13.8.2.x), c_ptr, or c_funptr (15.2.2)."
```

The editor doesn't think that this is quite right on technical grounds, let alone editorial, so it needs further development.

### 1 13.8.3.6 ERROR\_UNIT

- 2 The value of the default integer scalar constant ERROR\_UNIT identifies the processor-dependent pre-
- 3 connected external unit used for the purpose of error reporting (9.4). This unit may be the same as
- 4 OUTPUT\_UNIT. The value shall not be -1.

**1 13.8.3.7 FILE\_STORAGE\_SIZE**

2 The value of the default integer scalar constant FILE\_STORAGE\_SIZE is the size expressed in bits of  
3 the file storage unit (9.2.4).

**4 13.8.3.8 INPUT\_UNIT**

5 The value of the default integer scalar constant INPUT\_UNIT identifies the same processor-dependent  
6 preconnected external unit as the one identified by an asterisk in a READ statement (9.4). The value  
7 shall not be -1.

**8 13.8.3.9 INT8, INT16, INT32, and INT64**

9 The values of these default integer scalar named constants shall be those of the kind type parameters that  
10 specify an INTEGER type whose storage size expressed in bits is 8, 16, 32, and 64 respectively. If, for  
11 any of these constants, the processor supports more than one kind of that size, it is processor-dependent  
12 which kind value is provided. If the processor supports no kind of a particular size, that constant shall  
13 be equal to -2 if the processor supports kinds of a larger size and -1 otherwise.

**14 13.8.3.10 IOSTAT\_END**

15 The value of the default integer scalar constant IOSTAT\_END is assigned to the variable specified in  
16 an IOSTAT= specifier (9.10.4) if an end-of-file condition occurs during execution of an input/output  
17 statement and no error condition occurs. This value shall be negative.

**18 13.8.3.11 IOSTAT\_EOR**

19 The value of the default integer scalar constant IOSTAT\_EOR is assigned to the variable specified in  
20 an IOSTAT= specifier (9.10.4) if an end-of-record condition occurs during execution of an input/output  
21 statement and no end-of-file or error condition occurs. This value shall be negative and different from  
22 the value of IOSTAT\_END.

**23 13.8.3.12 IOSTAT\_INQUIRE\_INTERNAL\_UNIT**

24 The value of the default integer scalar constant IOSTAT\_INQUIRE\_INTERNAL\_UNIT is assigned to  
25 the variable specified in an IOSTAT= specifier in an INQUIRE statement (9.9) if a *file-unit-number*  
26 identifies an internal unit in that statement.

**NOTE 13.26**

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.
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**27 13.8.3.13 NUMERIC\_STORAGE\_SIZE**

28 The value of the default integer scalar constant NUMERIC\_STORAGE\_SIZE is the size expressed in  
29 bits of the numeric storage unit (16.5.3.2).

**30 13.8.3.14 OUTPUT\_UNIT**

31 The value of the default integer scalar constant OUTPUT\_UNIT identifies the same processor-dependent  
32 preconnected external unit as the one identified by an asterisk in a WRITE statement (9.4). The value  
33 shall not be -1.

1 **13.8.3.15 REAL32, REAL64, and REAL128**

2 The values of these default integer scalar named constants shall be those of the kind type parameters  
3 that specify a REAL type whose storage size expressed in bits is 32, 64, and 128 respectively. If, for  
4 any of these constants, the processor supports more than one kind of that size, it is processor-dependent  
5 which kind value is provided. If the processor supports no kind of a particular size, that constant shall  
6 be equal to  $-2$  if the processor supports kinds of a larger size and  $-1$  otherwise.

7 **13.8.4 The ISO\_C\_BINDING module**

8 The ISO\_C\_BINDING intrinsic module is described in Clause 15.



## 1 14 Exceptions and IEEE arithmetic

2 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.  
3  
4 If the module IEEE\_FEATURES is provided, which of the named constants defined in this standard are included is processor dependent. The module IEEE\_ARITHMETIC behaves as if it contained a  
5  
6 USE statement for IEEE\_EXCEPTIONS; everything that is public in IEEE\_EXCEPTIONS is public in  
7 IEEE\_ARITHMETIC.

### NOTE 14.1

The types and procedures defined in these modules are not themselves intrinsic.

8 If IEEE\_EXCEPTIONS or IEEE\_ARITHMETIC is accessible in a scoping unit, IEEE\_OVERFLOW  
9 and IEEE\_DIVIDE\_BY\_ZERO are supported in the scoping unit for all kinds of real and complex  
10 data. Which other exceptions are supported can be determined by the function IEEE\_SUPPORT\_  
11 FLAG (14.10.26); whether control of halting is supported can be determined by the function IEEE\_SUP-  
12 PORT\_HALTING. The extent of support of the other exceptions may be influenced by the accessibility  
13 of the named constants IEEE\_INEXACT\_FLAG, IEEE\_INVALID\_FLAG, and IEEE\_UNDERFLOW\_  
14 FLAG of the module IEEE\_FEATURES. If a scoping unit has access to IEEE\_UNDERFLOW\_FLAG  
15 of IEEE\_FEATURES, within the scoping unit the processor shall support underflow and return true  
16 from IEEE\_SUPPORT\_FLAG( IEEE\_UNDERFLOW, X) for at least one kind of real. Similarly, if  
17 IEEE\_INEXACT\_FLAG or IEEE\_INVALID\_FLAG is accessible, within the scoping unit the processor  
18 shall support the exception and return true from the corresponding inquiry for at least one kind of real.  
19 Also, if IEEE\_HALTING is accessible, within the scoping unit the processor shall support control of  
20 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  
 6 may be influenced by the accessibility of the named constants of the module IEEE\_FEATURES. If a  
 7 scoping unit has access to IEEE\_DATATYPE of IEEE\_FEATURES, within the scoping unit the pro-  
 8 cessor shall support IEEE arithmetic and return true from IEEE\_SUPPORT\_DATATYPE(X) (14.10.23)  
 9 for at least one kind of real. Similarly, if IEEE\_DENORMAL, IEEE\_DIVIDE, IEEE\_INF, IEEE\_NAN,  
 10 IEEE\_ROUNDING, or IEEE\_SQRT is accessible, within the scoping unit the processor shall support the  
 11 feature 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.

24 The module IEEE\_EXCEPTIONS defines the following types.

- 25       • IEEE\_FLAG\_TYPE is for identifying a particular exception flag. Its only possible values are those  
 26       of named constants defined in the module: IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_  
 27       BY\_ZERO, IEEE\_UNDERFLOW, and IEEE\_INEXACT. The module also defines the array named  
 28       constants IEEE\_USUAL = (/ IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_INVALID /)  
 29       and IEEE\_ALL = (/ IEEE\_USUAL, IEEE\_UNDERFLOW, IEEE\_INEXACT /).
- 30       • IEEE\_STATUS\_TYPE is for saving the current floating point status.

31 The module IEEE\_ARITHMETIC defines the following.

- 32       • The type IEEE\_CLASS\_TYPE, for identifying a class of floating-point values. Its only possible  
 33       values are those of named constants defined in the module: IEEE\_SIGNALING\_NAN, IEEE\_QUI-  
 34       ET\_NAN, IEEE\_NEGATIVE\_INF, IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_DENORM-  
 35       AL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO, IEEE\_POSITIVE\_DENORMAL, IEEE\_  
 36       POSITIVE\_NORMAL, IEEE\_POSITIVE\_INF, and IEEE\_OTHER\_VALUE.
- 37       • The type IEEE\_ROUND\_TYPE, for identifying a particular rounding mode. Its only possible  
 38       values are those of named constants defined in the module: IEEE\_NEAREST, IEEE\_TO\_ZERO,  
 39       IEEE\_UP, and IEEE\_DOWN for the IEEE modes; and IEEE\_OTHER for any other mode.

- 1     • The elemental operator `==` for two values of one of these types to return true if the values are the  
2       same and false otherwise.
- 3     • The elemental operator `/=` for two values of one of these types to return true if the values differ  
4       and false otherwise.

5 The module `IEEE_FEATURES` defines the type `IEEE_FEATURES_TYPE`, for expressing the  
6 need for particular IEEE features. Its only possible values are those of named constants de-  
7 fined in the module: `IEEE_DATATYPE`, `IEEE_DENORMAL`, `IEEE_DIVIDE`, `IEEE_HALTING`,  
8 `IEEE_INEXACT_FLAG`, `IEEE_INF`, `IEEE_INVALID_FLAG`, `IEEE_NAN`, `IEEE_ROUNDING`,  
9 `IEEE_SQRT`, and `IEEE_UNDERFLOW_FLAG`.

## 10 14.2 The exceptions

11 The exceptions are the following.

- 12     • `IEEE_OVERFLOW` occurs when the result for an intrinsic real operation or assignment has an  
13       absolute value greater than a processor-dependent limit, or the real or imaginary part of the result  
14       for an intrinsic complex operation or assignment has an absolute value greater than a processor-  
15       dependent limit.
- 16     • `IEEE_DIVIDE_BY_ZERO` occurs when a real or complex division has a nonzero numerator and a  
17       zero denominator.
- 18     • `IEEE_INVALID` occurs when a real or complex operation or assignment is invalid; possible examples  
19       are `SQRT(X)` when `X` is real and has a nonzero negative value, and conversion to an integer (by  
20       assignment, an intrinsic procedure, or a procedure defined in an intrinsic module) when the result  
21       is too large to be representable.
- 22     • `IEEE_UNDERFLOW` occurs when the result for an intrinsic real operation or assignment has an  
23       absolute value less than a processor-dependent limit and loss of accuracy is detected, or the real  
24       or imaginary part of the result for an intrinsic complex operation or assignment has an absolute  
25       value less than a processor-dependent limit and loss of accuracy is detected.
- 26     • `IEEE_INEXACT` occurs when the result of a real or complex operation or assignment is not exact.

27 Each exception has a flag whose value is either quiet or signaling. The value can be determined by  
28 the function `IEEE_GET_FLAG`. Its initial value is quiet and it signals when the associated excep-  
29 tion occurs. Its status can also be changed by the subroutine `IEEE_SET_FLAG` or the subroutine  
30 `IEEE_SET_STATUS`. Once signaling within a procedure, it remains signaling unless set quiet by an  
31 invocation of the subroutine `IEEE_SET_FLAG` or the subroutine `IEEE_SET_STATUS`.

32 If a flag is signaling on entry to a procedure other than `IEEE_GET_FLAG` or `IEEE_GET_STATUS`, the  
33 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.

34 Evaluation of a specification expression may cause an exception to signal.

35 In a scoping unit that has access to `IEEE_EXCEPTIONS` or `IEEE_ARITHMETIC`, if an intrinsic  
36 procedure or a procedure defined in an intrinsic module executes normally, the values of the flags  
37 `IEEE_OVERFLOW`, `IEEE_DIVIDE_BY_ZERO`, and `IEEE_INVALID` shall be as on entry to the proce-  
38 dure, even if one or more signals during the calculation. If a real or complex result is too large for the

1 procedure to handle, IEEE.OVERFLOW may signal. If a real or complex result is a NaN because of an  
 2 invalid operation (for example, LOG(-1.0)), IEEE.INVALID may signal. Similar rules apply to format  
 3 processing and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set  
 4 signaling because of an intermediate calculation that does not affect the result.

5 In a sequence of statements that has no invocations of IEEE.GET\_FLAG, IEEE.SET\_FLAG, IEEE.-  
 6 GET\_STATUS, IEEE.SET\_HALTING\_MODE, or IEEE.SET\_STATUS, if the execution of an operation  
 7 would cause an exception to signal but after execution of the sequence no value of a variable depends  
 8 on the operation, whether the exception is signaling is processor dependent. For example, when Y has  
 9 the value zero, whether the code

```
10      X = 1.0/Y
11      X = 3.0
```

12 signals IEEE.DIVIDE\_BY\_ZERO is processor dependent. Another example is the following:

```
13      REAL, PARAMETER :: X=0.0, Y=6.0
14      IF (1.0/X == Y) PRINT *, 'Hello world'
```

15 where the processor is permitted to discard the IF statement because the logical expression can never  
 16 be true and no value of a variable depends on it.

17 An exception shall not signal if this could arise only during execution of an operation beyond those  
 18 required or permitted by the standard. For example, the statement

```
19      IF (F(X)>0.0) Y = 1.0/Z
```

20 shall not signal IEEE.DIVIDE\_BY\_ZERO when both F(X) and Z are zero and the statement

```
21      WHERE(A>0.0) A = 1.0/A
```

22 shall not signal IEEE.DIVIDE\_BY\_ZERO. On the other hand, when X has the value 1.0 and Y has the  
 23 value 0.0, the expression

```
24      X>0.00001 .OR. X/Y>0.00001
```

25 is permitted to cause the signaling of IEEE.DIVIDE\_BY\_ZERO.

26 The processor need not support IEEE.INVALID, IEEE.UNDERFLOW, and IEEE.INEXACT. If an  
 27 exception is not supported, its flag is always quiet. The function IEEE.SUPPORT\_FLAG can be used  
 28 to inquire whether a particular flag is supported.

## 29 14.3 The rounding modes

30 The IEEE International Standard specifies four rounding modes.

- 31 • IEEE.NEAREST rounds the exact result to the nearest representable value.



- 1     • IEEE\_TO\_ZERO rounds the exact result towards zero to the next representable value.
  - 2     • IEEE\_UP rounds the exact result towards +infinity to the next representable value.
  - 3     • IEEE\_DOWN rounds the exact result towards -infinity to the next representable value.
- 4   The function IEEE\_GET\_ROUNDING\_MODE can be used to inquire which rounding mode is in operation. Its value is one of the above four or IEEE\_OTHER if the rounding mode does not conform to the IEEE International Standard.
- 7   If the processor supports the alteration of the rounding mode during execution, the subroutine IEEE\_SET\_ROUNDING\_MODE can be used to alter it. The function IEEE\_SUPPORT\_ROUNDING can be used to inquire whether this facility is available for a particular mode. The function IEEE\_SUPPORT\_IO can be used to inquire whether rounding for base conversion in formatted input/output (9.4.6.4, 9.5.1.12, 10.7.2.3.7) is as specified in the IEEE International Standard.
- 12   In a procedure other than IEEE\_SET\_ROUNDING\_MODE or IEEE\_SET\_STATUS, the processor shall not change the rounding mode on entry, and on return shall ensure that the rounding mode is the same 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.

## 15   14.4   Underflow mode

- 16   Some processors allow control during program execution of whether underflow produces a denormalized number in conformance with the IEEE International Standard (gradual underflow) or produces zero instead (abrupt underflow). On some processors, floating-point performance is typically better in abrupt underflow mode than in gradual underflow mode.
- 20   Control over the underflow mode is exercised by invocation of IEEE\_SET\_UNDERFLOW\_MODE. The function IEEE\_GET\_UNDERFLOW\_MODE can be used to inquire which underflow mode is in operation. The function IEEE\_SUPPORT\_UNDERFLOW\_MODE can be used to inquire whether this facility is available. The initial underflow mode is processor dependent. In a procedure other than IEEE\_SET\_UNDERFLOW\_MODE or IEEE\_SET\_STATUS, the processor shall not change the underflow mode on entry, and on return shall ensure that the underflow mode is the same as it was on entry.
- 26   The underflow mode affects only floating-point calculations whose type is that of an X for which IEEE\_SUPPORT\_UNDERFLOW\_CONTROL returns true.

## 28   14.5   Halting

- 29   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.

## 1 14.6 The floating point status

2 The values of all the supported flags for exceptions, rounding mode, underflow mode, and halting are  
 3 called the floating point status. The floating point status can be saved in a scalar variable of type  
 4 `TYPE(IEEE_STATUS_TYPE)` with the subroutine `IEEE_GET_STATUS` and restored with the subrou-  
 5 tine `IEEE_SET_STATUS`. There are no facilities for finding the values of particular flags held within such  
 6 a variable. Portions of the floating point status can be saved with the subroutines `IEEE_GET_FLAG`,  
 7 `IEEE_GET_HALTING_MODE`, and `IEEE_GET_ROUNDING_MODE`, and set with the subroutines  
 8 `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.

## 9 14.7 Exceptional values

10 The IEEE International Standard specifies the following exceptional floating point values.

- 11 • Denormalized values have very small absolute values and lowered precision.
- 12 • Infinite values (+infinity and -infinity) are created by overflow or division by zero.
- 13 • Not-a-Number ( NaN) values are undefined values or values created by an invalid operation.

14 In this standard, the term **normal** is used for values that are not in one of these exceptional classes.

15 The functions `IEEE_IS_FINITE`, `IEEE_IS_NAN`, `IEEE_IS_NEGATIVE`, and `IEEE_IS_NORMAL` are pro-  
 16 vided to test whether a value is finite, NaN, negative, or normal. The function `IEEE_VALUE` is pro-  
 17 vided to generate an IEEE number of any class, including an infinity or a NaN. The functions `IEEE_-`  
 18 `SUPPORT_DENORMAL`, `IEEE_SUPPORT_INF`, and `IEEE_SUPPORT_NAN` are provided to determine  
 19 whether these facilities are available for a particular kind of real.

## 20 14.8 IEEE arithmetic

21 The function `IEEE_SUPPORT_DATATYPE` can be used to inquire whether IEEE arithmetic is sup-  
 22 ported for a particular kind of real. Complete conformance with the IEEE International Standard is not  
 23 required, but the normalized numbers shall be exactly those of an IEEE floating-point format; the oper-  
 24 ations of addition, subtraction, and multiplication shall conform with at least one of the IEEE rounding  
 25 modes; the IEEE operation `rem` shall be provided by the function `IEEE_REM`; and the IEEE functions  
 26 `copysign`, `scalb`, `logb`, `nextafter`, and `unordered` shall be provided by the functions `IEEE_COPY_SIGN`,  
 27 `IEEE_SCALB`, `IEEE_LOGB`, `IEEE_NEXT_AFTER`, and `IEEE_UNORDERED`, respectively. The in-  
 28 quiry function `IEEE_SUPPORT_DIVIDE` is provided to inquire whether the processor supports divide  
 29 with the accuracy specified by the IEEE International Standard. For each of the operations of addi-  
 30 tion, subtraction, and multiplication, the result shall be as specified in the IEEE International Standard  
 31 whenever the IEEE result is normalized and the operands are normalized (if floating point) or are valid  
 32 and within range (if another type).

1 The inquiry function IEEE\_SUPPORT\_NAN is provided to inquire whether the processor supports  
 2 IEEE NaNs. Where these are supported, their behavior for unary and binary operations, including  
 3 those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the  
 4 specifications in the IEEE International Standard.

5 The inquiry function IEEE\_SUPPORT\_INF is provided to inquire whether the processor supports IEEE  
 6 infinities. Where these are supported, their behavior for unary and binary operations, including those  
 7 defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifi-  
 8 cations in the IEEE International Standard.

9 The inquiry function IEEE\_SUPPORT\_DENORMAL is provided to inquire whether the processor sup-  
 10 ports IEEE denormals. Where these are supported, their behavior for unary and binary operations,  
 11 including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent  
 12 with the specifications in the IEEE International Standard.

13 The IEEE International Standard specifies a square root function that returns -0.0 for the square root  
 14 of -0.0 and has certain accuracy requirements. The function IEEE\_SUPPORT\_SQRT can be used to  
 15 inquire whether SQRT conforms to the IEEE International Standard for a particular kind of real.

16 The inquiry function IEEE\_SUPPORT\_STANDARD is provided to inquire whether the processor sup-  
 17 ports all the IEEE facilities defined in this standard for a particular kind of real.

## 18 14.9 Tables of the procedures

19 For all of the procedures defined in the modules, the arguments shown are the names that shall be used  
 20 for argument keywords if the keyword form is used for the actual arguments.

21 The procedure classification terms “inquiry function” and “transformational function” are used here  
 22 with the same meanings as in 13.1.

### 23 14.9.1 Inquiry functions

24 The module IEEE\_EXCEPTIONS contains the following inquiry functions.

25	IEEE_SUPPORT_FLAG (FLAG [, X])	Are IEEE exceptions supported?
26	IEEE_SUPPORT_HALTING (FLAG)	Is IEEE halting control supported?

27 The module IEEE\_ARITHMETIC contains the following inquiry functions.

28	IEEE_SUPPORT_DATATYPE ([X])	Is IEEE arithmetic supported?
29	IEEE_SUPPORT_DENORMAL ([X])	Are IEEE denormalized numbers supported?
30	IEEE_SUPPORT_DIVIDE ([X])	Is IEEE divide supported?
31	IEEE_SUPPORT_INF ([X])	Is IEEE infinity supported?
32	IEEE_SUPPORT_IO ([X])	Is IEEE formatting supported?
33	IEEE_SUPPORT_NAN ([X])	Are IEEE NaNs supported?
	IEEE_SUPPORT_ROUNDING	Is IEEE rounding supported?
34	(ROUND_VALUE [, X])	
35	IEEE_SUPPORT_SQRT ([X])	Is IEEE square root supported?
36	IEEE_SUPPORT_STANDARD ([X])	Are all IEEE facilities supported?
37	IEEE_SUPPORT_UNDERFLOW_CONTROL ([X])	Is IEEE underflow control supported?

## 1 14.9.2 Elemental functions

2 The module IEEE\_ARITHMETIC contains the following elemental functions for reals X and Y for which  
3 IEEE\_SUPPORT\_DATATYPE(X) and IEEE\_SUPPORT\_DATATYPE(Y) are true.

4	IEEE_CLASS (X)	IEEE class.
5	IEEE_COPY_SIGN (X,Y)	IEEE copysign function.
6	IEEE_IS_FINITE (X)	Determine if value is finite.
7	IEEE_IS_NAN (X)	Determine if value is IEEE Not-a-Number.
8	IEEE_IS_NORMAL (X)	Determine if a value is normal, that is, neither an 9 infinity, a NaN, nor denormalized.
10	IEEE_IS_NEGATIVE (X)	Determine if value is negative.
11	IEEE_LOGB (X)	Unbiased exponent in the IEEE floating point 12 format.
13	IEEE_NEXT_AFTER (X,Y)	Returns the next representable neighbor of X in 14 the direction toward Y.
15	IEEE_REM (X,Y)	The IEEE REM function, that is $X - Y*N$ , where 16 N is the integer nearest to the exact value 17 $X/Y$ .
18	IEEE_RINT (X)	Round to an integer value according to the 19 current rounding mode.
20	IEEE_SCALB (X,I)	Returns $X \times 2^I$ .
21	IEEE_UNORDERED (X,Y)	IEEE unordered function. True if X or Y is a 22 NaN and false otherwise.
23	IEEE_VALUE (X,CLASS)	Generate an IEEE value.

## 24 14.9.3 Kind function

25 The module IEEE\_ARITHMETIC contains the following transformational function.

26	IEEE_SELECTED_REAL_KIND ([P, R, 27 RADIX])	Kind type parameter value for an IEEE real with given precision, range, and radix.
----	-----------------------------------------------	---------------------------------------------------------------------------------------

## 28 14.9.4 Elemental subroutines

29 The module IEEE\_EXCEPTIONS contains the following elemental subroutines.

30	IEEE_GET_FLAG (FLAG,FLAG_VALUE)	Get an exception flag.
31	IEEE_GET_HALTING_MODE (FLAG, HALTING)	Get halting mode for an exception.

## 32 14.9.5 Nonelemental subroutines

33 The module IEEE\_EXCEPTIONS contains the following nonelemental subroutines.

34	IEEE_GET_STATUS (STATUS_VALUE)	Get the current state of the floating point 35 environment.
36	IEEE_SET_FLAG (FLAG,FLAG_VALUE)	Set an exception flag.
37	IEEE_SET_HALTING_MODE (FLAG, HALTING)	Controls continuation or halting on exceptions.
38	IEEE_SET_STATUS (STATUS_VALUE)	Restore the state of the floating point 39 environment.

1 The nonelemental subroutines IEEE\_SET\_FLAG and IEEE\_SET\_HALTING\_MODE are pure. No other  
 2 nonelemental subroutine contained in IEEE\_EXCEPTIONS is pure.

3 The module IEEE\_ARITHMETIC contains the following nonelemental subroutines.

4	IEEE_GET_ROUNDING_MODE (ROUND_VALUE)	Get the current IEEE rounding mode.
5	IEEE_GET_UNDERFLOW_MODE (GRADUAL)	Get the current underflow mode.
6	IEEE_SET_ROUNDING_MODE (ROUND_VALUE)	Set the current IEEE rounding mode.
7	IEEE_SET_UNDERFLOW_MODE (GRADUAL)	Set the current underflow mode.

8 No nonelemental subroutine contained in IEEE\_ARITHMETIC is pure.

## 9 14.10 Specifications of the procedures

10 In the detailed descriptions below, procedure names are generic and are not specific. All the functions  
 11 are pure. The dummy arguments of the intrinsic module procedures in 14.9.1, 14.9.2, and 14.9.3 have  
 12 INTENT(IN). The dummy arguments of the intrinsic module procedures in 14.9.4 and 14.9.5 have  
 13 INTENT(IN) if the intent is not stated explicitly. In the examples, it is assumed that the processor  
 14 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.

15 For the elemental functions of IEEE\_ARITHMETIC, as tabulated in 14.9.2, if X or Y has a value that  
 16 is an infinity or a NaN, the result shall be consistent with the general rules in 6.1 and 6.2 of the IEEE  
 17 International Standard. For example, the result for an infinity shall be constructed as the limiting case  
 18 of the result with a value of arbitrarily large magnitude, if such a limit exists.

### 19 14.10.1 IEEE\_CLASS (X)

20 **Description.** IEEE class function.

21 **Class.** Elemental function.

22 **Argument.** X shall be of type real.

23 **Restriction.** IEEE\_CLASS(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) has  
 24 the value false.

**Result Characteristics.** TYPE(IEEE\_CLASS\_TYPE).

**Result Value.** The result value shall be IEEE\_SIGNALING\_NAN or IEEE\_QUIET\_NAN if IEEE\_SUPPORT\_NAN(X) has the value true and the value of X is a signaling or quiet NaN, respectively. The result value shall be IEEE\_NEGATIVE\_INF or IEEE\_POSITIVE\_INF if IEEE\_SUPPORT\_INF(X) has the value true and the value of X is negative or positive infinity, respectively. The result value shall be IEEE\_NEGATIVE\_DENORMAL or IEEE\_POSITIVE\_DENORMAL if IEEE\_SUPPORT\_DENORMAL(X) has the value true and the value of X is a negative or positive denormalized value, respectively. The result value shall be IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO, or IEEE\_POSITIVE\_NORMAL if value of X is negative normal, negative zero, positive zero, or positive normal, respectively. Otherwise, the result value shall be IEEE\_OTHER\_VALUE.

**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.

### 14.10.2 IEEE\_COPY\_SIGN (X, Y)

**Description.** IEEE copysign function.

**Class.** Elemental function.

**Arguments.** The arguments shall be of type real.

**Restriction.** IEEE\_COPY\_SIGN(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATA\_TYPE(X) or IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

**Result Characteristics.** Same as X.

**Result Value.** The result has the value of X with the sign of Y. This is true even for IEEE special values, such as a NaN or an infinity (on processors supporting such values).

**Example.** The value of IEEE\_COPY\_SIGN(X,1.0) is ABS(X) even when X is NaN.

### 14.10.3 IEEE\_GET\_FLAG (FLAG, FLAG\_VALUE)

**Description.** Get an exception flag.

**Class.** Elemental subroutine.

**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\_

1       VALUE is true if the IEEE\_OVERFLOW flag is signaling and is false if it is quiet.

## 2   **14.10.4   IEEE\_GET\_HALTING\_MODE (FLAG, HALTING)**

3       **Description.** Get halting mode for an exception.

4       **Class.** Elemental subroutine.

5       **Arguments.**

6       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.

7       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.

8       **Example.** To store the halting mode for IEEE\_OVERFLOW, do a calculation without halting, and restore the halting mode later:

```
10       USE, INTRINSIC :: IEEE_ARITHMETIC
11       LOGICAL HALTING
12       ...
13       CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Store halting mode
14       CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.FALSE.) ! No halting
15       ...! calculation without halting
16       CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Restore halting mode
```

## 17   **14.10.5   IEEE\_GET\_ROUNDING\_MODE (ROUND\_VALUE)**

18       **Description.** Get the current IEEE rounding mode.

19       **Class.** Subroutine.

20       **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.

24       **Example.** To store the rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```
26       USE, INTRINSIC :: IEEE_ARITHMETIC
27       TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
28       ...
29       CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
30       CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
31       ... ! calculation with round to nearest
32       CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode
```

## 33   **14.10.6   IEEE\_GET\_STATUS (STATUS\_VALUE)**

1       **Description.** Get the current value of the floating point status (14.6).

2       **Class.** Subroutine.

3       **Argument.** STATUS\_VALUE shall be scalar of type TYPE(IEEE\_STATUS\_TYPE). It is an  
4       INTENT(OUT) argument and returns the floating point status.

5       **Example.** To store all the exception flags, do a calculation involving exception handling, and  
6       restore them later:

```

7       USE, INTRINSIC :: IEEE_ARITHMETIC
8       TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
9       ...
10      CALL IEEE_GET_STATUS(STATUS_VALUE) ! Get the flags
11      CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set the flags quiet.
12      ... ! calculation involving exception handling
13      CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags

```

#### 14   **14.10.7   IEEE\_GET\_UNDERFLOW\_MODE (GRADUAL)**

15       **Description.** Get the current underflow mode (14.4).

16       **Class.** Subroutine.

17       **Argument.** GRADUAL shall be scalar and of type default logical. It is an INTENT(OUT)  
18       argument. The value is true if the current underflow mode is gradual underflow, and false if the  
19       current underflow mode is abrupt underflow.

20       **Restriction.** IEEE\_GET\_UNDERFLOW\_MODE shall not be invoked unless IEEE\_SUPPORT-  
21       \_UNDERFLOW\_CONTROL(X) is true for some X.

22       **Example.** After CALL IEEE\_SET\_UNDERFLOW\_MODE(.FALSE.), a subsequent CALL  
23       IEEE\_GET\_UNDERFLOW\_MODE(GRADUAL) will set GRADUAL to false.

#### 24   **14.10.8   IEEE\_IS\_FINITE (X)**

25       **Description.** Determine if a value is finite.

26       **Class.** Elemental function.

27       **Argument.** X shall be of type real.

28       **Restriction.** IEEE\_IS\_FINITE(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X)  
29       has the value false.

30       **Result Characteristics.** Default logical.

31       **Result Value.** The result has the value true if the value of X is finite, that is, IEEE\_CLASS(X)  
32       has one of the values IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_DENORMAL, IEEE-  
33       NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO, IEEE\_POSITIVE\_DENORMAL, or IEEE\_POS-  
34       ITIVE\_NORMAL; otherwise, the result has the value false.

35       **Example.** IEEE\_IS\_FINITE(1.0) has the value true.

#### 36   **14.10.9   IEEE\_IS\_NAN (X)**



1       **Description.** Determine if a value is IEEE Not-a-Number.

2       **Class.** Elemental function.

3       **Argument.** X shall be of type real.

4       **Restriction.** IEEE\_IS\_NAN(X) shall not be invoked if IEEE\_SUPPORT\_NAN(X) has the value  
5 false.

6       **Result Characteristics.** Default logical.

7       **Result Value.** The result has the value true if the value of X is an IEEE NaN; otherwise, it  
8 has the value false.

9       **Example.** IEEE\_IS\_NAN(SQRT(-1.0)) has the value true if IEEE\_SUPPORT\_SQRT(1.0) has  
10 the value true.

#### 11   **14.10.10 IEEE\_IS\_NEGATIVE (X)**

12       **Description.** Determine if a value is negative.

13       **Class.** Elemental function.

14       **Argument.** X shall be of type real.

15       **Restriction.** IEEE\_IS\_NEGATIVE(X) shall not be invoked if IEEE\_SUPPORT\_DATA-  
16 TYPE(X) has the value false.

17       **Result Characteristics.** Default logical.

18       **Result Value.** The result has the value true if IEEE\_CLASS(X) has one of the values  
19 IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_DENORMAL, IEEE\_NEGATIVE\_ZERO or  
20 IEEE\_NEGATIVE\_INF; otherwise, the result has the value false.

21       **Example.** IEEE\_IS\_NEGATIVE(0.0)) has the value false.

#### 22   **14.10.11 IEEE\_IS\_NORMAL (X)**

23       **Description.** Determine if a value is normal, that is, neither an infinity, a NaN, nor denormal-  
24 ized.

25       **Class.** Elemental function.

26       **Argument.** X shall be of type real.

27       **Restriction.** IEEE\_IS\_NORMAL(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X)  
28 has the value false.

29       **Result Characteristics.** Default logical.

30       **Result Value.** The result has the value true if IEEE\_CLASS(X) has one of the values  
31 IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO or IEEE\_-  
32 POSITIVE\_NORMAL; otherwise, the result has the value false.

33       **Example.** IEEE\_IS\_NORMAL(SQRT(-1.0)) has the value false if IEEE\_SUPPORT\_SQRT(1.0)  
34 has the value true.

#### 35   **14.10.12 IEEE\_LOGB (X)**

**Description.** Unbiased exponent in the IEEE floating point format.

**Class.** Elemental function.

**Argument.** X shall be of type real.

**Restriction.** IEEE\_LOGB(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) has the value false.

**Result Characteristics.** Same as X.

**Result Value.**

*Case (i):* If the value of X is neither zero, infinity, nor NaN, the result has the value of the unbiased exponent of X. Note: this value is equal to EXPONENT(X)-1.

*Case (ii):* If X==0, the result is -infinity if IEEE\_SUPPORT\_INF(X) is true and -HUGE(X) otherwise; IEEE\_DIVIDE\_BY\_ZERO signals.

**Example.** IEEE\_LOGB(-1.1) has the value 0.0.

### 14.10.13 IEEE\_NEXT\_AFTER (X, Y)

**Description.** Returns the next representable neighbor of X in the direction toward Y.

**Class.** Elemental function.

**Arguments.** The arguments shall be of type real.

**Restriction.** IEEE\_NEXT\_AFTER(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) or IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

**Result Characteristics.** Same as X.

**Result Value.**

*Case (i):* If X == Y, the result is X and no exception is signaled.

*Case (ii):* If X /= Y, the result has the value of the next representable neighbor of X in the direction of Y. The neighbors of zero (of either sign) are both nonzero. IEEE\_OVERFLOW is signaled when X is finite but IEEE\_NEXT\_AFTER(X,Y) is infinite; IEEE\_UNDERFLOW is signaled when IEEE\_NEXT\_AFTER(X,Y) is denormalized; in both cases, IEEE\_INEXACT signals.

**Example.** The value of IEEE\_NEXT\_AFTER(1.0,2.0) is 1.0+EPSILON(X).

### 14.10.14 IEEE\_REM (X, Y)

**Description.** IEEE REM function.

**Class.** Elemental function.

**Arguments.** The arguments shall be of type real.

**Restriction.** IEEE\_REM(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) or IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

**Result Characteristics.** Real with the kind type parameter of whichever argument has the greater precision.

**Result Value.** The result value, regardless of the rounding mode, shall be exactly  $X - Y*N$ ,

where  $N$  is the integer nearest to the exact value  $X/Y$ ; whenever  $|N - X/Y| = 1/2$ ,  $N$  shall be even. If the result value is zero, the sign shall be that of  $X$ .

**Examples.** The value of `IEEE_REM(4.0,3.0)` is 1.0, the value of `IEEE_REM(3.0,2.0)` is -1.0, and the value of `IEEE_REM(5.0,2.0)` is 1.0.

#### 14.10.15 IEEE\_RINT (X)

**Description.** Round to an integer value according to the current rounding mode.

**Class.** Elemental function.

**Argument.**  $X$  shall be of type real.

**Restriction.** `IEEE_RINT(X)` shall not be invoked if `IEEE_SUPPORT_DATATYPE(X)` has the value false.

**Result Characteristics.** Same as  $X$ .

**Result Value.** The value of the result is the value of  $X$  rounded to an integer according to the current rounding mode. If the result has the value zero, the sign is that of  $X$ .

**Examples.** If the current rounding mode is round to nearest, the value of `IEEE_RINT(1.1)` is 1.0. If the current rounding mode is round up, the value of `IEEE_RINT(1.1)` is 2.0.

#### 14.10.16 IEEE\_SCALB (X, I)

**Description.** Returns  $X \times 2^I$ .

**Class.** Elemental function.

**Arguments.**

$X$  shall be of type real.

$I$  shall be of type integer.

**Restriction.** `IEEE_SCALB(X)` shall not be invoked if `IEEE_SUPPORT_DATATYPE(X)` has the value false.

**Result Characteristics.** Same as  $X$ .

**Result Value.**

*Case (i):* If  $X \times 2^I$  is representable as a normal number, the result has this value.

*Case (ii):* If  $X$  is finite and  $X \times 2^I$  is too large, the `IEEE_OVERFLOW` exception shall occur. If `IEEE_SUPPORT_INF(X)` is true, the result value is infinity with the sign of  $X$ ; otherwise, the result value is `SIGN(HUGE(X),X)`.

*Case (iii):* If  $X \times 2^I$  is too small and there is loss of accuracy, the `IEEE_UNDERFLOW` exception shall occur. The result is the representable number having a magnitude nearest to  $|2^I|$  and the same sign as  $X$ .

*Case (iv):* If  $X$  is infinite, the result is the same as  $X$ ; no exception signals.

**Example.** The value of `IEEE_SCALB(1.0,2)` is 4.0.

#### 14.10.17 IEEE\_SELECTED\_REAL\_KIND ([P, R, RADIX])

**Description.** Returns a value of the kind type parameter of an IEEE real data type with decimal precision of at least P digits, a decimal exponent range of at least R, and a radix of RADIX. For data objects of such a type, IEEE\_SUPPORT\_DATATYPE(X) has the value true.

**Class.** Transformational function.

**Arguments.** At least one argument shall be present.

P (optional) shall be scalar and of type integer.

R (optional) shall be scalar and of type integer.

RADIX (optional) shall be scalar and of type integer.

**Result Characteristics.** Default integer scalar.

**Result Value.** If P or R is absent, the result value is the same as if it were present with the value zero. If RADIX is absent, there is no requirement on the radix of the selected kind. The result has a value equal to a value of the kind type parameter of an IEEE real type with decimal precision, as returned by the function PRECISION, of at least P digits, a decimal exponent range, as returned by the function RANGE, of at least R, and a radix, as returned by the function RADIX, of RADIX, if such a kind type parameter is available on the processor.

Otherwise, the result is  $-1$  if the processor supports an IEEE real type with radix RADIX and exponent range of at least R but not with precision of at least P,  $-2$  if the processor supports an IEEE real type with radix RADIX and precision of at least P but not with exponent range of at least R,  $-3$  if the processor supports an IEEE real type with radix RADIX but with neither precision of at least P nor exponent range of at least R,  $-4$  if the processor supports an IEEE real type with radix RADIX and either precision of at least P or exponent range of at least R but not both together, and  $-5$  if the processor supports no IEEE real type with radix RADIX.

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

**Example.** IEEE\_SELECTED\_REAL\_KIND(6,30) has the value KIND(0.0) on a machine that supports IEEE single precision arithmetic for its default real approximation method.

#### 14.10.18 IEEE\_SET\_FLAG (FLAG, FLAG\_VALUE)

**Description.** Assign a value to an exception flag.

**Class.** Pure subroutine.

**Arguments.**

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\_UNDERFLOW, or IEEE\_INEXACT, the corresponding exception flag is assigned a value. No two elements of FLAG shall have the same value.

FLAG\_VALUE shall be a 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.

**Example.** CALL IEEE\_SET\_FLAG(IEEE\_OVERFLOW,.TRUE.) sets the IEEE\_OVERFLOW

1 flag to be signaling.

## 2 14.10.19 IEEE\_SET\_HALTING\_MODE (FLAG, HALTING)

3 **Description.** Controls continuation or halting after an exception.

4 **Class.** Pure subroutine.

5 **Arguments.**

6 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.

7 HALTING shall be a scalar or array of type default logical. It shall be conformable with FLAG. If an element has the value is true, the corresponding exception specified by FLAG will cause halting. Otherwise, execution will continue after this exception.

8 **Restriction.** IEEE\_SET\_HALTING\_MODE(FLAG,HALTING) shall not be invoked if IEEE\_SUPPORT\_HALTING(FLAG) has the value false.

10 **Example.** CALL IEEE\_SET\_HALTING\_MODE(IEEE\_DIVIDE\_BY\_ZERO,.TRUE.) causes  
11 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.

## 12 14.10.20 IEEE\_SET\_ROUNDING\_MODE (ROUND\_VALUE)

13 **Description.** Set the current IEEE rounding mode.

14 **Class.** Subroutine.

15 **Argument.** ROUND\_VALUE shall be scalar and of type TYPE(IEEE\_ROUND\_TYPE). It  
16 specifies the mode to be set.

17 **Restriction.** IEEE\_SET\_ROUNDING\_MODE(ROUND\_VALUE) shall not be invoked un-  
18 less IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE,X) is true for some X such that IEEE-  
19 SUPPORT\_DATATYPE(X) is true.

20 **Example.** To store the rounding mode, do a calculation with round to nearest, and restore the  
21 rounding mode later:

```
22 USE, INTRINSIC :: IEEE_ARITHMETIC
23 TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
24 ...
25 CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
26 CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
27 : ! calculation with round to nearest
28 CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode
```

## 29 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  
4       value shall 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  
6       restore 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  
18       underflow mode is set to gradual underflow. If it is false, the current underflow mode is set to  
19       abrupt underflow.

20       **Restriction.** IEEE\_SET\_UNDERFLOW\_MODE shall not be invoked unless IEEE\_SUPPORT-  
21       \_UNDERFLOW\_CONTROL(X) is true for some X.

22       **Example.** To perform some calculations with abrupt underflow and then restore the previous  
23       mode:

```

24     USE,INTRINSIC :: IEEE_ARITHMETIC
25     LOGICAL SAVE_UNDERFLOW_MODE
26     ...
27     CALL IEEE_GET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)
28     CALL IEEE_SET_UNDERFLOW_MODE(GRADUAL=.FALSE.)
29     ... ! Perform some calculations with abrupt underflow
30     CALL IEEE_SET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)

```

### 31 14.10.23 IEEE\_SUPPORT\_DATATYPE () or IEEE\_SUPPORT\_DATATYPE (X)

32       **Description.** Inquire whether the processor supports IEEE arithmetic.

33       **Class.** Inquiry function.

1       **Argument.** X shall be of type real. It may be a scalar or an array.

2       **Result Characteristics.** Default logical scalar.

3       **Result Value.** The result has the value true if the processor supports IEEE arithmetic for all  
4       reals (X absent) or for real variables of the same kind type parameter as X; otherwise, it has  
5       the value false. Here, support is as defined in the first paragraph of 14.8.

6       **Example.** If default real type conforms to the IEEE International Standard except that un-  
7       derflow values flush to zero instead of being denormal, IEEE\_SUPPORT\_DATATYPE(1.0) has  
8       the value true.

#### 9   **14.10.24 IEEE\_SUPPORT\_DENORMAL () or IEEE\_SUPPORT\_DENORMAL (X)**

10       **Description.** Inquire whether the processor supports IEEE denormalized numbers.

11       **Class.** Inquiry function.

12       **Argument.** X shall be of type real. It may be a scalar or an array.

13       **Result Characteristics.** Default logical scalar.

14       **Result Value.**

15       *Case (i):*       IEEE\_SUPPORT\_DENORMAL(X) has the value true if IEEE\_SUPPORT\_-  
16       DATATYPE(X) has the value true and the processor supports arithmetic op-  
17       erations and assignments with denormalized numbers (biased exponent  $e = 0$   
18       and fraction  $f \neq 0$ , see subclause 3.2 of the IEEE International Standard) for  
19       real variables of the same kind type parameter as X; otherwise, it has the value  
20       false.

21       *Case (ii):*     IEEE\_SUPPORT\_DENORMAL() has the value true if and only if IEEE\_SUP-  
22       PORT\_DENORMAL(X) has the value true for all real X.

23       **Example.** IEEE\_SUPPORT\_DENORMAL(X) has the value true if the processor supports  
24       denormalized numbers for X.

#### **NOTE 14.12**

The denormalized numbers are not included in the 13.4 model for real numbers; they satisfy the inequality  $ABS(X) < TINY(X)$ . They usually occur as a result of an arithmetic operation whose exact result is less than  $TINY(X)$ . Such an operation causes IEEE\_UNDERFLOW to signal unless the result is exact. IEEE\_SUPPORT\_DENORMAL(X) is false if the processor never returns a denormalized number as the result of an arithmetic operation.

#### 25   **14.10.25 IEEE\_SUPPORT\_DIVIDE () or IEEE\_SUPPORT\_DIVIDE (X)**

26       **Description.** Inquire whether the processor supports divide with the accuracy specified by the  
27       IEEE International Standard.

28       **Class.** Inquiry function.

29       **Argument.** X shall be of type real. It may be a scalar or an array.

30       **Result Characteristics.** Default logical scalar.

31       **Result Value.**

32       *Case (i):*       IEEE\_SUPPORT\_DIVIDE(X) has the value true if the processor supports divide  
33       with the accuracy specified by the IEEE International Standard for real variables

1 of the same kind type parameter as X; otherwise, it has the value false.  
 2 *Case (ii):* IEEE\_SUPPORT\_DIVIDE() has the value true if and only if IEEE\_SUPPORT\_  
 3 DIVIDE(X) has the value true for all real X.

4 **Example.** IEEE\_SUPPORT\_DIVIDE(X) has the value true if the processor supports IEEE  
 5 divide for X.

#### 6 14.10.26 IEEE\_SUPPORT\_FLAG (FLAG) or IEEE\_SUPPORT\_FLAG (FLAG, X)

7 **Description.** Inquire whether the processor supports an exception.

8 **Class.** Inquiry function.

9 **Arguments.**

10 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\_UNDERFLOW, or IEEE\_INEXACT.

11 X shall be of type real. It may be a scalar or an array.

12 **Result Characteristics.** Default logical scalar.

13 **Result Value.**

14 *Case (i):* IEEE\_SUPPORT\_FLAG(FLAG, X) has the value true if the processor supports  
 15 detection of the specified exception for real variables of the same kind type pa-  
 16 rameter as X; otherwise, it has the value false.

17 *Case (ii):* IEEE\_SUPPORT\_FLAG(FLAG) has the value true if and only if IEEE\_SUP-  
 18 PORT\_FLAG(FLAG, X) has the value true for all real X.

19 **Example.** IEEE\_SUPPORT\_FLAG(IEEE\_INEXACT) has the value true if the processor sup-  
 20 ports the inexact exception.

#### 21 14.10.27 IEEE\_SUPPORT\_HALTING (FLAG)

22 **Description.** Inquire whether the processor supports the ability to control during program  
 23 execution whether to abort or continue execution after an exception.

24 **Class.** Inquiry function.

25 **Argument.** FLAG shall be scalar and of type TYPE(IEEE\_FLAG\_TYPE). Its value shall be  
 26 one of IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_UNDERFLOW,  
 27 or IEEE\_INEXACT.

28 **Result Characteristics.** Default logical scalar.

29 **Result Value.** The result has the value true if the processor supports the ability to control  
 30 during program execution whether to abort or continue execution after the exception specified  
 31 by FLAG; otherwise, it has the value false. Support includes the ability to change the mode by  
 32 CALL IEEE\_SET\_HALTING(FLAG).

33 **Example.** IEEE\_SUPPORT\_HALTING(IEEE\_OVERFLOW) has the value true if the proces-  
 34 sor supports control of halting after an overflow.

#### 35 14.10.28 IEEE\_SUPPORT\_INF () or IEEE\_SUPPORT\_INF (X)

36 **Description.** Inquire whether the processor supports the IEEE infinity facility.



**Class.** Inquiry function.

**Argument.** X shall be of type real. It may be a scalar or an array.

**Result Characteristics.** Default logical scalar.

**Result Value.**

*Case (i):* IEEE\_SUPPORT\_INF(X) has the value true if the processor supports IEEE infinities (positive and negative) for real variables of the same kind type parameter as X; otherwise, it has the value false.

*Case (ii):* IEEE\_SUPPORT\_INF() has the value true if and only if IEEE\_SUPPORT\_INF(X) has the value true for all real X.

**Example.** IEEE\_SUPPORT\_INF(X) has the value true if the processor supports IEEE infinities for X.

## 14.10.29 IEEE\_SUPPORT\_IO () or IEEE\_SUPPORT\_IO (X)

**Description.** Inquire whether the processor supports IEEE base conversion rounding during formatted input/output (9.4.6.4, 9.5.1.12, 10.7.2.3.7).

**Class.** Inquiry function.

**Argument.** X shall be of type real. It may be a scalar or an array.

**Result Characteristics.** Default logical scalar.

**Result Value.**

*Case (i):* IEEE\_SUPPORT\_IO(X) has the value true if the processor supports IEEE base conversion during formatted input/output (9.4.6.4, 9.5.1.12, 10.7.2.3.7) as described in the IEEE International Standard for the modes UP, DOWN, ZERO, and NEAREST for real variables of the same kind type parameter as X; otherwise, it has the value false.

*Case (ii):* IEEE\_SUPPORT\_IO() has the value true if and only if IEEE\_SUPPORT\_IO(X) has the value true for all real X.

**Example.** IEEE\_SUPPORT\_IO(X) has the value true if the processor supports IEEE base conversion for X.

## 14.10.30 IEEE\_SUPPORT\_NAN () or IEEE\_SUPPORT\_NAN (X)

**Description.** Inquire whether the processor supports the IEEE Not-a-Number facility.

**Class.** Inquiry function.

**Argument.** X shall be of type real. It may be a scalar or an array.

**Result Characteristics.** Default logical scalar.

**Result Value.**

*Case (i):* IEEE\_SUPPORT\_NAN(X) has the value true if the processor supports IEEE NaNs for real variables of the same kind type parameter as X; otherwise, it has the value false.

*Case (ii):* IEEE\_SUPPORT\_NAN() has the value true if and only if IEEE\_SUPPORT\_NAN(X) has the value true for all real X.

**Example.** IEEE\_SUPPORT\_NAN(X) has the value true if the processor supports IEEE NaNs for X.

### 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.

**Class.** Inquiry function.

**Arguments.**

ROUND\_VALUE shall be of type TYPE(IEEE\_ROUND\_TYPE).

X shall be of type real. It may be a scalar or an array.

**Result Characteristics.** Default logical scalar.

**Result Value.**

*Case (i):* IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE, X) has the value true if the processor supports the rounding mode defined by ROUND\_VALUE for real variables of the same kind type parameter as X; otherwise, it has the value false. Support includes the ability to change the mode by CALL IEEE\_SET\_ROUNDING\_MODE(ROUND\_VALUE).

*Case (ii):* IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE) has the value true if and only if IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE, X) has the value true for all real X.

**Example.** IEEE\_SUPPORT\_ROUNDING(IEEE\_TO\_ZERO) has the value true if the processor supports rounding to zero for all reals.

### 14.10.32 IEEE\_SUPPORT\_SQRT () or IEEE\_SUPPORT\_SQRT (X)

**Description.** Inquire whether the intrinsic function SQRT conforms to the IEEE International Standard.

**Class.** Inquiry function.

**Argument.** X shall be of type real. It may be a scalar or an array.

**Result Characteristics.** Default logical scalar.

**Result Value.**

*Case (i):* IEEE\_SUPPORT\_SQRT(X) has the value true if the intrinsic function SQRT conforms to the IEEE International Standard for real variables of the same kind type parameter as X; otherwise, it has the value false.

*Case (ii):* IEEE\_SUPPORT\_SQRT() has the value true if and only if IEEE\_SUPPORT\_SQRT(X) has the value true for all real X.

**Example.** If IEEE\_SUPPORT\_SQRT(1.0) has the value true, SQRT(−0.0) will have the value −0.0.

### 14.10.33 IEEE\_SUPPORT\_STANDARD () or IEEE\_SUPPORT\_STANDARD (X)

**Description.** Inquire whether the processor supports all the IEEE facilities defined in this standard.

**Class.** Inquiry function.

**Argument.** X shall be of type real. It may be a scalar or an array.

**Result Characteristics.** Default logical scalar.

**Result Value.**

*Case (i):* IEEE\_SUPPORT\_STANDARD(X) has the value true if the results of all the functions IEEE\_SUPPORT\_DATATYPE(X), IEEE\_SUPPORT\_DENORMAL(X), IEEE\_SUPPORT\_DIVIDE(X), IEEE\_SUPPORT\_FLAG(FLAG,X) for valid FLAG, IEEE\_SUPPORT\_HALTING(FLAG) for valid FLAG, IEEE\_SUPPORT\_INF(X), IEEE\_SUPPORT\_NAN(X), IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE,X) for valid ROUND\_VALUE, and IEEE\_SUPPORT\_SQRT(X) are all true; otherwise, the result has the value false.

*Case (ii):* IEEE\_SUPPORT\_STANDARD() has the value true if and only if IEEE\_SUPPORT\_STANDARD(X) has the value true for all real X.

**Example.** IEEE\_SUPPORT\_STANDARD() has the value false if the processor supports both IEEE and non-IEEE kinds of reals.

#### 14.10.34 IEEE\_SUPPORT\_UNDERFLOW\_CONTROL() or IEEE\_SUPPORT\_UNDERFLOW\_CONTROL(X)

**Description.** Inquire whether the procedure supports the ability to control the underflow mode during program execution.

**Class.** Inquiry function.

**Argument.** X shall be of type real. It may be a scalar or an array.

**Result Characteristics.** Default logical scalar.

**Result Value.**

*Case (i):* IEEE\_SUPPORT\_UNDERFLOW\_CONTROL(X) has the value true if the processor supports control of the underflow mode for floating-point calculations with the same type as X, and false otherwise.

*Case (ii):* IEEE\_SUPPORT\_UNDERFLOW\_CONTROL() has the value true if the processor supports control of the underflow mode for all floating-point calculations, and false otherwise.

**Example.** IEEE\_SUPPORT\_UNDERFLOW\_CONTROL(2.5) has the value true if the processor supports underflow mode control for calculations of type default real.

#### 14.10.35 IEEE\_UNORDERED (X, Y)

**Description.** IEEE unordered function. True if X or Y is a NaN, and false otherwise.

**Class.** Elemental function.

**Arguments.** The arguments shall be of type real.

**Restriction.** IEEE\_UNORDERED(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) or IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

**Result Characteristics.** Default logical.

**Result Value.** The result has the value true if X or Y is a NaN or both are NaNs; otherwise, it has the value false.

**Example.** IEEE.UNORDERED(0.0,SQRT(-1.0)) has the value true 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\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO or IEEE\_POSITIVE\_NORMAL.

### 14.10.36 IEEE\_VALUE (X, CLASS)

**Description.** Generate an IEEE value.

**Class.** Elemental function.

**Arguments.**

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\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO or IEEE\_POSITIVE\_NORMAL.

**Restriction.** IEEE\_VALUE(X,CLASS) shall not be invoked if IEEE\_SUPPORT\_DATA\_TYPE(X) has the value false.

**Result Characteristics.** Same as X.

**Result Value.** The result value is an IEEE value as specified by CLASS. Although in most cases the value is processor dependent, the value shall not vary between invocations for any particular X kind type parameter and CLASS value.

**Example.** IEEE\_VALUE(1.0,IEEE\_NEGATIVE\_INF) has the value -infinity.

## 14.11 Examples

### NOTE 14.13

```

MODULE DOT
! Module for dot product of two real arrays of rank 1.
! The caller needs to ensure that exceptions do not cause halting.
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL :: MATRIX_ERROR = .FALSE.
INTERFACE OPERATOR(.dot.)
MODULE PROCEDURE MULT
END INTERFACE
CONTAINS
REAL FUNCTION MULT(A,B)
REAL, INTENT(IN) :: A(:),B(:)
INTEGER I
LOGICAL OVERFLOW

```

**NOTE 14.13 (cont.)**

```

      IF (SIZE(A)/=SIZE(B)) THEN
        MATRIX_ERROR = .TRUE.
        RETURN
      END IF
! The processor ensures that IEEE_OVERFLOW is quiet
      MULT = 0.0
      DO I = 1, SIZE(A)
        MULT = MULT + A(I)*B(I)
      END DO
      CALL IEEE_GET_FLAG(IEEE_OVERFLOW,OVERFLOW)
      IF (OVERFLOW) MATRIX_ERROR = .TRUE.
    END FUNCTION MULT
  END MODULE DOT

```

This module provides the dot product of two real arrays of rank 1. If the sizes of the arrays are different, an immediate return occurs with MATRIX\_ERROR true. If overflow occurs during the actual calculation, the IEEE\_OVERFLOW flag will signal and MATRIX\_ERROR will be true.

**NOTE 14.14**

```

USE, INTRINSIC :: IEEE_EXCEPTIONS
USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_INVALID_FLAG
! The other exceptions of IEEE_USUAL (IEEE_OVERFLOW and
! IEEE_DIVIDE_BY_ZERO) are always available with IEEE_EXCEPTIONS
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
LOGICAL, DIMENSION(3) :: FLAG_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE)
CALL IEEE_SET_HALTING_MODE(IEEE_USUAL,.FALSE.) ! Needed in case the
!           default on the processor is to halt on exceptions
CALL IEEE_SET_FLAG(IEEE_USUAL,.FALSE.)
! First try the "fast" algorithm for inverting a matrix:
MATRIX1 = FAST_INV(MATRIX) ! This shall not alter MATRIX.
CALL IEEE_GET_FLAG(IEEE_USUAL,FLAG_VALUE)
IF (ANY(FLAG_VALUE)) THEN
! "Fast" algorithm failed; try "slow" one:
  CALL IEEE_SET_FLAG(IEEE_USUAL,.FALSE.)
  MATRIX1 = SLOW_INV(MATRIX)
  CALL IEEE_GET_FLAG(IEEE_USUAL,FLAG_VALUE)
  IF (ANY(FLAG_VALUE)) THEN
    WRITE (*, *) 'Cannot invert matrix'
    STOP
  END IF
END IF
END IF

```

**NOTE 14.14 (cont.)**

```
CALL IEEE_SET_STATUS(STATUS_VALUE)
```

In this example, the function FAST\_INV may cause a condition to signal. If it does, another try is made with SLOW\_INV. If this still fails, a message is printed and the program stops. Note, also, that it is important to set the flags quiet before the second try. The state of all the flags is stored and restored.

**NOTE 14.15**

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL FLAG_VALUE
...
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.FALSE.)
! First try a fast algorithm for inverting a matrix.
CALL IEEE_SET_FLAG(IEEE_OVERFLOW,.FALSE.)
DO K = 1, N
...
    CALL IEEE_GET_FLAG(IEEE_OVERFLOW,FLAG_VALUE)
    IF (FLAG_VALUE) EXIT
END DO
IF (FLAG_VALUE) THEN
! Alternative code which knows that K-1 steps have executed normally.
...
END IF
```

Here the code for matrix inversion is in line and the transfer is made more precise by adding extra tests of the flag.

**NOTE 14.16**

```
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 )
```

## NOTE 14.16 (cont.)

```

CALL IEEE_GET_FLAG(OUT_OF_RANGE, FLAGS)
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) ) ! may 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\_INTPTR\_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\_UINT shall be a valid value for a bits kind type parameter of the processor. The val-  
 4 ues of C\_USHORT, C\_ULONG, C\_ULONG\_LONG, C\_UNSIGNED\_CHAR, C\_UINT8\_T, C\_UINT16\_T,  
 5 C\_UINT32\_T, C\_UINT64\_T, C\_UINT\_LEAST8\_T, C\_UINT\_LEAST16\_T, C\_UINT\_LEAST32\_T, C\_UINT\_-  
 6 LEAST64\_T, C\_UINT\_FAST8\_T, C\_UINT\_FAST16\_T, C\_UINT\_FAST32\_T, C\_UINT\_FAST64\_T, C\_UINTMAX\_-  
 7 T, C\_UINTPTR\_T, C\_FLOAT\_BITS, C\_DOUBLE\_BITS, C\_LONG\_DOUBLE\_BITS, C\_FLOAT\_COMPLEX\_-  
 8 BITS, C\_DOUBLE\_COMPLEX\_BITS, C\_LONG\_DOUBLE\_COMPLEX\_BITS, and C\_BOOL\_BITS shall  
 9 each be a valid value for a bits kind type parameter on the processor, -1 if the companion C processor  
 10 defines the corresponding C type and there is no interoperating Fortran processor kind, or -2 if the C  
 11 processor does not define the corresponding C type.

#### J3 internal note

##### Unresolved Technical Issue 74

This has taken all the “obvious” names for constants that interoperate with unsigned integers, should we consider adding proper unsigned integer support in a future Fortran. And all for the dubious pleasure of saving a few keystrokes from BITS\_KIND(0\_C\_SHORT) et al, when one wants to declare BITS arguments or components.

But since BITS actuals are compatible with INTEGER dummies, that seems like an almost complete waste of namespace.

As for C\_FLOAT\_COMPLEX\_BITS, instead of BITS\_KIND(C\_FLOAT\_COMPLEX)...

Furthermore, the names are completely misleading, because they interoperate equally well with signed as well as unsigned *just like the existing C\_INT* et al. The right name for these is C\_INT\_BITS et al, if we need them at all.

- 12 The value of C\_CHAR shall be a valid value for a character kind type parameter on the processor or  
 13 shall be -1. The value of C\_CHAR is known as the **C character kind**.
- 14 The following entities shall be named constants of type character with a length parameter of one. The  
 15 kind parameter value shall be equal to the value of C\_CHAR unless C\_CHAR = -1, in which case the  
 16 kind parameter value shall be the same as for default kind. The values of these constants are specified  
 17 in Table 15.1. In the case that C\_CHAR  $\neq$  -1 the value is specified using C syntax. The semantics of  
 18 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).

- 1 The entities `C_PTR` and `C_FUNPTR` are described in 15.3.3.
- 2 The entity `C_NULL_PTR` shall be a named constant of type `C_PTR`. The value of `C_NULL_PTR` shall
- 3 be the same as the value `NULL` in C. The entity `C_NULL_FUNPTR` shall be a named constant of type
- 4 `C_FUNPTR`. The value of `C_NULL_FUNPTR` shall be that of a null pointer to a function in C.

### 5 **15.2.3 Procedures in the module**

- 6 In the detailed descriptions below, procedure names are generic and not specific.

- 7 A C procedure argument is often defined in terms of a **C address**. The `C_LOC` and `C_FUNLOC`
- 8 functions are provided so that Fortran applications can determine the appropriate value to use with C
- 9 facilities. The `C_ASSOCIATED` function is provided so that Fortran programs can compare C addresses.
- 10 The `C_F_POINTER` and `C_F_PROCPOINTER` subroutines provide a means of associating a Fortran
- 11 pointer with the target of a C pointer.

#### 12 **15.2.3.1 C\_ASSOCIATED (C\_PTR\_1 [, C\_PTR\_2])**

- 13 **Description.** Indicates the association status of `C_PTR_1` or indicates whether `C_PTR_1` and
- 14 `C_PTR_2` are associated with the same entity.

- 15 **Class.** Inquiry function.

- 16 **Arguments.**

- 17 `C_PTR_1` shall be a scalar of type `C_PTR` or `C_FUNPTR`.

- 18 `C_PTR_2` shall be a scalar of the same type as `C_PTR_1`.
- (optional)

- 19 **Result Characteristics.** Default logical scalar.

- 20 **Result Value.**

- 21 *Case (i):* If `C_PTR_2` is absent, the result is false if `C_PTR_1` is a C null pointer and true
- 22 otherwise.

- 23 *Case (ii):* If `C_PTR_2` is present, the result is false if `C_PTR_1` is a C null pointer. Otherwise,
- 24 the result is true if `C_PTR_1` compares equal to `C_PTR_2` in the sense of 6.3.2.3
- 25 and 6.5.9 of 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
```

**NOTE 15.4 (cont.)**

```

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)

```

**1 15.2.3.2 C\_F\_POINTER (CPTR, FPTR [, SHAPE])**

2 **Description.** Associates a data pointer with the target of a C pointer and specifies its shape.

3 **Class.** Subroutine.

4 **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.

6 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. 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.

7 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.  
(optional) SHAPE shall be present if and only if FPTR is an array; its size shall be equal to the rank of FPTR.

**9 15.2.3.3 C\_F\_PROCPINTER (CPTR, FPTR)**

10 **Description.** Associates a procedure pointer with the target of a C function pointer.

11 **Class.** Subroutine.

12 **Arguments.**

13 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. 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.

#### 2 15.2.3.4 C\_FUNLOC (X)

3 **Description.** Returns the C address of the argument.

4 **Class.** Inquiry function.

5 **Argument.** X shall either be a procedure that is interoperable, or a procedure pointer associated with an interoperable procedure.

7 **Result Characteristics.** Scalar of type C\_FUNPTR.

8 **Result Value.**

9 The result value is described using the result name CPTR. The result is determined as if  
10 C\_FUNPTR were a derived type containing an implicit-interface procedure pointer component  
11 PX and the pointer assignment CPTR%PX => X were executed.

12 The result is a value that can be used as an actual CPTR argument in a call to C\_F\_PROC-  
13 POINTER where FPTR has attributes that would allow the pointer assignment FPTR => X.  
14 Such a call to C\_F\_PROCPOINTER shall have the effect of the pointer assignment FPTR =>  
15 X.

#### 16 15.2.3.5 C\_LOC (X)

17 **Description.** Returns the C address of the argument.

18 **Class.** Inquiry function.

19 **Argument.** X shall either

- 20 (1) have interoperable type and type parameters and be
- 21 (a) a variable that has the TARGET attribute and is interoperable,
  - 22 (b) an allocated allocatable variable that has the TARGET attribute and is not an array  
23 of zero size, or
  - 24 (c) an associated scalar pointer, or
- 25 (2) be a nonpolymorphic scalar, have no length type parameters, and be
- 26 (a) a nonallocatable, nonpointer variable that has the TARGET attribute,
  - 27 (b) an allocated allocatable variable that has the TARGET attribute, or
  - 28 (c) an associated pointer.

29 **Result Characteristics.** Scalar of type C\_PTR.

30 **Result Value.**

31 The result value is described using the result name CPTR.

32 If X is a scalar data entity, the result is determined as if C\_PTR were a derived type containing a scalar

1 pointer component PX of the type and type parameters of X and the pointer assignment CPTR%PX  
2 => X were executed.

3 If X is an array data entity, the result is determined as if C\_PTR were a derived type containing a scalar  
4 pointer component PX of the type and type parameters of X and the pointer assignment of CPTR%PX  
5 to the first element of X were executed.

6 If X is a data entity that is interoperable or has interoperable type and type parameters, the  
7 result is the value that the C processor returns as the result of applying the unary "&" operator  
8 (as defined in the C International Standard, 6.5.3.2) to the target of CPTR%PX.

9 The result is a value that can be used as an actual CPTR argument in a call to C\_F\_POINTER  
10 where FPTR has attributes that would allow the pointer assignment FPTR => X. Such a call  
11 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).

#### 12 15.2.3.6 C\_SIZEOF (X)

13 **Description.** Returns the size of X in bytes.

14 **Class.** Inquiry function.

15 **Argument.** X shall be an interoperable data entity that is not an assumed-size array.

16 **Result Characteristics.** Scalar integer of kind C\_SIZE\_T (15.3.2).

17 **Result Value.**

18 If X is scalar, the result value is the value that the C processor returns as the result of applying  
19 the sizeof operator (C International Standard, subclause 6.5.3.4) to an object of a type that  
20 interoperates with the type and type parameters of X.

21 If X is an array, the result value is the value that the C processor returns as the result of applying  
22 the sizeof operator to an object of a type that interoperates with the type and type parameters of X,  
23 multiplied by the number of elements in X.

### 24 15.3 Interoperability between Fortran and C entities

#### 25 15.3.1 General

26 Subclause 15.3 define the conditions under which a Fortran entity is interoperable. If a Fortran entity  
27 is interoperable, an equivalent entity may be defined by means of C and the Fortran entity is said to be  
28 interoperable 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.

### 1 15.3.2 Interoperability of intrinsic types

2 Table 15.2 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic  
 3 type with particular type parameter values is interoperable with a C type if the type and kind type  
 4 parameter value are listed in the table on the same row as that C type; if the type is character, inter-  
 5 operability also requires that the length type parameter be omitted or be specified by an initialization  
 6 expression whose value is one. A combination of Fortran type and type parameters that is interoperable  
 7 with a C type listed in the table is also interoperable with any unqualified C type that is compatible  
 8 with the listed C type.

9 The second column of the table refers to the named constants made accessible by the ISO\_C\_BINDING  
 10 intrinsic module. If the value of any of these named constants is negative, there is no combination of  
 11 Fortran type and type parameters interoperable with the C type shown in that row.

12 A combination of intrinsic type and type parameters is **interoperable** if it is interoperable with a C  
 13 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
INTEGER	C_INT	int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char unsigned char
	C_SIZE_T	size_t
	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
	C_INTMAX_T	intmax_t
	C_INTPTR_T	intptr_t
REAL	C_FLOAT	float
	C_DOUBLE	double
	C_LONG_DOUBLE	long double
	C_FLOAT_COMPLEX	float _Complex

## Interoperability between Fortran and C types

(cont.)

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
COMPLEX	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
long long	C_UINT	unsigned int or int
	C_USHORT	unsigned short or short
	C_ULONG	unsigned long or long
	C_ULONG_LONG	unsigned long long
signed char	C_UNSIGNED_CHAR	unsigned char
	C_UINT8_T	uint8_t or int8_t
int_least8_t	C_UINT16_T	uint16_t or int16_t
	C_UINT32_T	uint32_t or int32_t
int_least16_t	C_UINT64_T	uint64_t or int64_t
	C_UINT_LEAST8_T	uint_least8_t
int_least32_t	C_UINT_LEAST16_T	uint_least16_t
	C_UINT_LEAST32_T	uint_least32_t
BITS	C_UINT_LEAST64_T	uint_least64_t
	C_UINT_FAST8_T	uint_fast8_t or int_fast8_t
int_fast16_t	C_UINT_FAST16_T	uint_fast16_t
	C_UINT_FAST32_T	uint_fast32_t
int_fast32_t	C_UINT_FAST64_T	uint_fast64_t
	C_UINTMAX_T	uintmax_t or intmax_t
int_fast64_t	C_UINTPTR_T	uintptr_t or intptr_t
	C_FLOAT_BITS	float
	C_DOUBLE_BITS	double
	C_LONG_DOUBLE_BITS	long double
	C_FLOAT_COMPLEX_BITS	float _Complex
	C_DOUBLE_COMPLEX_BITS	double _Complex
	C_LONG_DOUBLE_COMPLEX_BITS	long double _Complex
	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. Because Fortran does not provide direct support for unsigned kinds of integers, the ISO\_C\_BINDING module does not make accessible named constants for their kind type parameter values. Instead 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. C\_PTR is interoperable  
3 with any C object pointer type. 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.

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
    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 sequence derived types are considered to be the same 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**

A polymorphic, allocatable, or pointer array is never interoperable. Such arrays are not 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 (12.5.1.8) 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 standard does not provide a mechanism for Fortran procedures to interoperate with such C functions.

19 A formal parameter of a C function prototype corresponds to a dummy argument of a Fortran interface if  
20 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
```

**NOTE 15.23 (cont.)**

```

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 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) :: &
&    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

```

**NOTE 15.24 (cont.)**

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 (12.5.1.8). 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/
  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).

### 31 15.5.2 Binding labels for procedures

32 The **binding label** of a procedure is a value of type default character that specifies the name by which  
33 a procedure with the BIND attribute is known to the companion processor.

34 If a procedure has the BIND attribute with the NAME= specifier and the value of its expression, after  
35 discarding leading and trailing blanks, has nonzero length, the procedure has this as its binding label.  
36 The case of letters in the binding label is significant. If a procedure has the BIND attribute with no  
37 NAME= specifier, and the procedure is not a dummy procedure or procedure pointer, then the binding  
38 label of the procedure is the same as the name of the procedure using lower case letters. Otherwise, the

- 1 procedure has no binding label.
- 2 C1506 A procedure defined in a submodule shall not have a binding label unless its interface is declared
- 3 in the ancestor module.
- 4 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.

### 5 15.5.3 Exceptions and IEEE arithmetic procedures

- 6 A procedure defined by means other than Fortran shall not use signal (C International Standard, 7.14.1)
- 7 to change the handling of any exception that is being handled by the Fortran processor.
- 8 A procedure defined by means other than Fortran shall not alter the floating point status (14.6) other
- 9 than by setting an exception flag to signaling.
- 10 The values of the floating point exception flags on entry to a procedure defined by means other than
- 11 Fortran are processor-dependent.



## 1 16 Scope, association, and definition

### 2 16.1 Identifiers and entities

3 Entities are identified by identifiers within a **scope** that is a program, a scoping unit, a construct, a  
4 single statement, or part of a statement.

- 5 • A **global identifier** has a scope of a program (2.2.1);
- 6 • A **local identifier** has a scope of a scoping unit (2.2);
- 7 • An identifier of a **construct entity** has a scope of a construct (7.4.3, 7.4.4, 8.1);
- 8 • An identifier of a **statement entity** has a scope of a statement or part of a statement (3.3).

9 An entity may be identified by

- 10 (1) a name (3.2.1),
- 11 (2) a statement label (3.2.4),
- 12 (3) an external input/output unit number (9.4),
- 13 (4) an identifier of a pending data transfer operation (9.5.1.8, 9.6),
- 14 (5) a submodule identifier (11.2.2),
- 15 (6) a generic identifier (12.4.2.1), or
- 16 (7) a binding label (15.5.2, 15.4.2).

17 By means of association, an entity may be referred to by the same identifier or a different identifier in  
18 a different scoping unit, or by a different identifier in the same scoping unit.

### 19 16.2 Scope of global identifiers

20 Program units, common blocks, external procedures, and entities with binding labels are **global entities**  
21 of a program. The name of a non-submodule program unit, common block, or external procedure is  
22 a global identifier and shall not be the same as the name of any other such global entity in the same  
23 program, except that an intrinsic module may have the same name as another program unit, common  
24 block, or external procedure in the same program. The submodule identifier of a submodule is a global  
25 identifier and shall not be the same as the submodule identifier of any other submodule. A binding  
26 label of an entity of the program is a global identifier and shall not be the same as the binding label of  
27 any other entity of the program; nor shall it be the same as the name of any other global entity of the  
28 program that is not an intrinsic module, ignoring differences in case. An entity of the program shall not  
29 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 12.4.2.1 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 (12.6.2.4). 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 Restrictions on generic declarations

25 This subclause contains the rules that shall be satisfied by every pair of specific procedures that have  
26 the same generic identifier within a scoping unit. If a generic procedure is accessed from a module, the

- 1 rules apply to all the specific versions even if some of them are inaccessible by their specific names.

**NOTE 16.6**

In most scoping units, the possible sources of procedures with a particular generic identifier are the accessible interface blocks and the generic bindings other than names for the accessible objects in that scoping unit. In a type definition, they are the generic bindings, including those from a parent type.

- 2 Two dummy arguments are **distinguishable** if neither is a subroutine and neither is TKR compatible  
 3 (4.3.1.3) with the other.
- 4 Within a scoping unit, if two procedures have the same generic operator and the same number of  
 5 arguments or both define assignment, one shall have a dummy argument that corresponds by position  
 6 in the argument list to a dummy argument of the other that is distinguishable with it.
- 7 Within a scoping unit, if two procedures have the same *dtio-generic-spec* (12.4.2.1), their *dtv* arguments  
 8 shall be type incompatible or have different kind type parameters.
- 9 Within a scoping unit, two procedures that have the same generic name shall both be subroutines or  
 10 both be functions, and
- 11 (1) there is a non-passed-object dummy data object in one or the other of them such that  
 12 (a) the number of dummy data objects in one that are nonoptional, are not passed-object,  
 13 and with which that dummy data object is TKR compatible, possibly including that  
 14 dummy data object itself,  
 15 exceeds  
 16 (b) the number of non-passed-object dummy data objects, both optional and nonoptional,  
 17 in the other that are not distinguishable with that dummy data object,
- 18 (2) both have passed-object dummy arguments and the passed-object dummy arguments are  
 19 distinguishable, or  
 20 (3) at least one of them shall have both  
 21 (a) a nonoptional non-passed-object dummy argument at an effective position such that  
 22 either the other procedure has no dummy argument at that effective position or the  
 23 dummy argument at that position is distinguishable with it, and  
 24 (b) a nonoptional non-passed-object dummy argument whose name is such that either the  
 25 other procedure has no dummy argument with that name or the dummy argument  
 26 with that name is distinguishable with it.  
 27 and the dummy argument that disambiguates by position shall either be the same as or  
 28 occur earlier in the argument list than the one that disambiguates by name.

- 29 The **effective position** of a dummy argument is its position in the argument list after any passed-object  
 30 dummy argument has been removed.

- 31 Within a scoping unit, if a generic name is the same as the name of a generic intrinsic procedure, the  
 32 generic intrinsic procedure is not accessible if the procedures in the interface and the intrinsic procedure  
 33 are not all functions or not all subroutines. If a generic invocation applies to both a specific procedure  
 34 from an interface and an accessible generic intrinsic procedure, it is the specific procedure from the  
 35 interface that is referenced.

**NOTE 16.7**

An extensive explanation of the application of these rules is in C.12.2.

### 1 16.3.5 Components, type parameters, and bindings

2 A component name has the scope of its derived-type definition. Outside the type definition, it may  
3 appear only within a designator of a component of a structure of that type or as a component keyword  
4 in a structure constructor for that type.

5 A type parameter name has the scope of its derived-type definition. Outside the derived-type definition,  
6 it may appear only as a type parameter keyword in a *derived-type-spec* for the type or as the *type-param-*  
7 *name* of a *type-param-inquiry*.

8 The binding name (4.5.5) of a type-bound procedure has the scope of its derived-type definition. Outside  
9 of the derived-type definition, it may appear only as the *binding-name* in a procedure reference.

10 A generic binding for which the *generic-spec* is not a *generic-name* has a scope that consists of all scoping  
11 units in which an entity of the type is accessible.

12 A component name or binding name may appear only in scoping units in which it is accessible.

13 The accessibility of components and bindings is specified in 4.5.4.7 and 4.5.5.

### 14 16.3.6 Argument keywords

15 As an argument keyword, a dummy argument name in an internal procedure, module procedure, or  
16 an interface body has a scope of the scoping unit of the host of the procedure or interface body. It  
17 may appear only in a procedure reference for the procedure of which it is a dummy argument. If the  
18 procedure or interface body is accessible in another scoping unit by use association or host association  
19 (16.5.1.3, 16.5.1.4), the argument keyword is accessible for procedure references for that procedure in  
20 that scoping unit.

21 A dummy argument name in an intrinsic procedure has a scope as an argument keyword of the scoping  
22 unit in which the reference to the procedure occurs. As an argument keyword, it may appear only in a  
23 procedure reference for the procedure of which it is a dummy argument.

## 24 16.4 Statement and construct entities

25 A variable that appears as a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array  
26 constructor, as a dummy argument in a statement function statement, or as an *index-name* in a FORALL  
27 statement is a statement entity. A variable that appears as an *index-name* in a FORALL or DO  
28 CONCURRENT construct or as an *associate-name* in a SELECT TYPE or ASSOCIATE construct is  
29 a construct entity. A macro local variable is a construct entity. An entity that is declared in a BLOCK  
30 construct and is not accessed by use association is a construct entity.

31 If a global or local identifier is the same as that of a construct entity, the identifier is interpreted within  
32 the construct as that of the construct entity. Elsewhere in the scoping unit, the identifier is interpreted  
33 as the global or local identifier.

34 If a global or local identifier accessible in the scoping unit of a statement is the same as the name of a  
35 statement entity in that statement, the name is interpreted within the scope of the statement entity as  
36 that of the statement entity. Elsewhere in the scoping unit, including parts of the statement outside the  
37 scope of the statement entity, the name is interpreted as the global or local identifier.

38 If the name of a statement entity is the same as the name of a construct entity and the statement is  
39 within the scope of the construct entity, the name is interpreted within the scope of the statement entity  
40 as that of the statement entity. Elsewhere in the construct, including parts of the statement outside the  
41 scope of the statement entity, the name is interpreted as that of the construct entity.

1 Except for a common block name or a scalar variable name, a global identifier or a local identifier of  
2 class (1) (16.3) in the scoping unit that contains a statement shall not be the name of a statement entity  
3 of that statement. Within the scope of a statement entity, another statement entity shall not have the  
4 same name.

5 The name of a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array constructor  
6 has a scope of its *data-implied-do* or *ac-implied-do*. It is a scalar variable that has the type and type  
7 parameters that it would have if it were the name of a variable in the scoping unit that includes the  
8 DATA statement or array constructor, and this type shall be integer type; it has no other attributes.  
9 The appearance of a name as a *data-i-do-variable* of an implied DO in a DATA statement or an *ac-do-*  
10 *variable* in an array constructor is not an implicit declaration of a variable whose scope is the scoping  
11 unit that contains the statement.

12 The name of a variable that appears as an *index-name* in a FORALL statement or FORALL or DO  
13 CONCURRENT construct has a scope of the statement or construct. It is a scalar variable that has the  
14 type and type parameters that it would have if it were the name of a variable in the scoping unit that  
15 includes the FORALL, and this type shall be integer type; it has no other attributes. The appearance  
16 of a name as an *index-name* in a FORALL statement or FORALL or DO CONCURRENT construct is  
17 not an implicit declaration of a variable whose scope is the scoping unit that contains the statement or  
18 construct.

19 The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement  
20 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  
21 in the scoping unit that includes the statement function; it has no other attributes.

22 Except for a common block name or a scalar variable name, a global identifier or a local identifier of class  
23 (1) (16.3) in the scoping unit of a FORALL statement, FORALL construct, or DO CONCURRENT  
24 construct shall not be the same as any of its *index-names*. An *index-name* of a contained FORALL  
25 statement, FORALL construct, or DO CONCURRENT construct shall not be the same as an *index-*  
26 *name* of any of its containing FORALL or DO CONCURRENT constructs.

27 The associate name of a SELECT TYPE construct has a separate scope for each block of the construct.  
28 Within each block, it has the declared type, dynamic type, type parameters, rank, and bounds specified  
29 in 8.1.6.2.

30 The associate names of an ASSOCIATE construct have the scope of the block. They have the declared  
31 type, dynamic type, type parameters, rank, and bounds specified in 8.1.5.2.

32 The macro local variables of a macro definition have the scope of the macro definition.

## 33 16.5 Association

### 34 16.5.1 Name association

#### 35 16.5.1.1 Forms of name association

36 There are five forms of **name association**: argument association, use association, host association,  
37 linkage association, and construct association. Argument, use, and host association provide mechanisms  
38 by which entities known in one scoping unit may be accessed in another scoping unit.

#### 39 16.5.1.2 Argument association

40 The rules governing argument association are given in Clause 12. As explained in 12.5, execution of a  
41 procedure reference establishes an association between an actual argument and its corresponding dummy

- 1 argument. Argument association may be sequence association (12.5.1.8).  
 2 The name of the dummy argument may be different from the name, if any, of its associated actual  
 3 argument. The dummy argument name is the name by which the associated actual argument is known,  
 4 and by which it may be accessed, in the referenced procedure.

**NOTE 16.8**

An actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.

- 5 Upon termination of execution of a procedure reference, all argument associations established by that  
 6 reference are terminated. A dummy argument of that procedure may be associated with an entirely  
 7 different actual argument in a subsequent invocation of the procedure.

**8 16.5.1.3 Use association**

- 9 **Use association** is the association of names in different scoping units specified by a USE statement. The  
 10 rules governing use association are given in 11.2.1. They allow for renaming of entities being accessed.  
 11 Use association allows access in one scoping unit to entities defined in another scoping unit; it remains  
 12 in effect throughout the execution of the program.

**13 16.5.1.4 Host association**

- 14 An internal subprogram, a module subprogram, a submodule subprogram, a module procedure interface  
 15 body, or a derived-type definition has access to entities from its host via **host association**. An interface  
 16 body that is not a module procedure interface body has access via host association to the named entities  
 17 from its host that are made accessible by IMPORT statements in the interface body. The accessed  
 18 entities are identified by the same identifier and have the same attributes as in the host, except that an  
 19 accessed entity may have the VOLATILE or ASYNCHRONOUS attribute even if the host entity does  
 20 not. The accessed entities are named data objects, derived types, abstract interfaces, module procedure  
 21 interfaces, procedures, generic identifiers, macros, and namelist groups.

- 22 If an entity that is accessed by use association has the same nongeneric name as a host entity, the host  
 23 entity is inaccessible by that name. Within the scoping unit, a name that is declared to be an external  
 24 procedure name by an *external-stmt*, *procedure-declaration-stmt*, or *interface-body*, or that appears as a  
 25 *module-name* in a *use-stmt* is a global identifier; any entity of the host that has this as its nongeneric  
 26 name is inaccessible by that name. A name that appears in the scoping unit as

- 27 (1) a *function-name* in a *stmt-function-stmt* or in an *entity-decl* in a *type-declaration-stmt*,  
 28 (2) an *object-name* in an *entity-decl* in a *type-declaration-stmt*, in a *pointer-stmt*, in a *save-stmt*,  
 29 in an *allocatable-stmt*, or in a *target-stmt*,  
 30 (3) a *type-param-name* in a *derived-type-stmt*,  
 31 (4) a *named-constant* in a *named-constant-def* in a *parameter-stmt*,  
 32 (5) an *array-name* in a *dimension-stmt*,  
 33 (6) a *variable-name* in a *common-block-object* in a *common-stmt*,  
 34 (7) a *proc-pointer-name* in a *common-block-object* in a *common-stmt*,  
 35 (8) the name of a variable that is wholly or partially initialized in a *data-stmt*,  
 36 (9) the name of an object that is wholly or partially equivalenced in an *equivalence-stmt*,  
 37 (10) a *dummy-arg-name* in a *function-stmt*, in a *subroutine-stmt*, in an *entry-stmt*, or in a *stmt-*  
 38 *function-stmt*,  
 39 (11) a *result-name* in a *function-stmt* or in an *entry-stmt*,  
 40 (12) the name of an entity declared by an interface body,

- 1 (13) an *intrinsic-procedure-name* in an *intrinsic-stmt*,
- 2 (14) a *namelist-group-name* in a *namelist-stmt*,
- 3 (15) the name of a macro in a *define-macro-stmt*,
- 4 (16) a *generic-name* in a *generic-spec* in an *interface-stmt*, or
- 5 (17) the name of a named construct

6 is a local identifier in the scoping unit and any entity of the host that has this as its nongeneric name is  
 7 inaccessible by that name by host association. If a scoping unit is the host of a derived-type definition  
 8 or a subprogram that does not define a separate module procedure, the name of the derived type or of  
 9 any procedure defined by the subprogram is a local identifier in the scoping unit; any entity of the host  
 10 that has this as its nongeneric name is inaccessible by that name. Local identifiers of a subprogram are  
 11 not accessible to its host.

#### NOTE 16.9

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.

12 If a host entity is inaccessible only because a local variable with the same name is wholly or partially  
 13 initialized in a DATA statement, the local variable shall not be referenced or defined prior to the DATA  
 14 statement.

15 If a derived-type name of a host is inaccessible, data entities of that type or subobjects of such data  
 16 entities still can be accessible.

#### NOTE 16.10

An interface body that is not a module procedure interface body accesses by host association only those entities made accessible by IMPORT statements.

17 If an external or dummy procedure with an implicit interface is accessed via host association, then it  
 18 shall have the EXTERNAL attribute in the host scoping unit; if it is invoked as a function in the inner  
 19 scoping unit, its type and type parameters shall be established in the host scoping unit. The type and  
 20 type parameters of a function with the EXTERNAL attribute are established in a scoping unit if that  
 21 scoping unit explicitly declares them, invokes the function, accesses the function from a module, or  
 22 accesses the function from its host where its type and type parameters are established.

23 If an intrinsic procedure is accessed via host association, then it shall be established to be intrinsic in the  
 24 host scoping unit. An intrinsic procedure is established to be intrinsic in a scoping unit if that scoping  
 25 unit explicitly gives it the INTRINSIC attribute, invokes it as an intrinsic procedure, accesses it from a  
 26 module, or accesses it from its host where it is established to be intrinsic.

#### NOTE 16.11

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
```



**NOTE 16.11 (cont.)**

```

PROGRAM A
USE B; USE C; USE D
...
CONTAINS
  SUBROUTINE INNER_PROC (Q)
    USE C           ! Not needed
    USE B, ONLY: BX ! Entities accessible are BX, IX, and JX
                    ! if no other IX or JX
                    ! is accessible to INNER_PROC
                    ! Q is local to INNER_PROC,
                    ! because Q is a dummy argument
    USE D, X => DX   ! Entities accessible are DX, DY, and DZ
                    ! X is local name for DX in INNER_PROC
                    ! X and DX denote same entity if no other
                    ! entity DX is local to INNER_PROC
    USE E, ONLY: EX ! EX is accessible in INNER_PROC, not in program A
                    ! EY and EZ are not accessible in INNER_PROC
                    ! or in program A
    USE G           ! FX and GX are accessible in INNER_PROC
    ...
  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 16.12**

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.

#### 1 16.5.1.6 Construct association

2 Execution of a SELECT TYPE statement establishes an association between the selector and the asso-  
3 ciate name of the construct. Execution of an ASSOCIATE statement establishes an association between  
4 each selector and the corresponding associate name of the construct.

5 If the selector is allocatable, it shall be allocated; the associate name is associated with the data object  
6 and does not have the ALLOCATABLE attribute.

7 If the selector has the POINTER attribute, it shall be associated; the associate name is associated with  
8 the target of the pointer and does not have the POINTER attribute.

9 If the selector is a variable other than an array section having a vector subscript, the association is  
10 with the data object specified by the selector; otherwise, the association is with the value of the selector  
11 expression, which is evaluated prior to execution of the block.

12 Each associate name remains associated with the corresponding selector throughout the execution of the  
13 executed block. Within the block, each selector is known by and may be accessed by the corresponding  
14 associate name. Upon termination of the construct, the association is terminated.

#### NOTE 16.13

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*.

### 15 16.5.2 Pointer association

#### 16 16.5.2.1 General

17 Pointer association between a pointer and a target allows the target to be referenced by a reference to the  
18 pointer. At different times during the execution of a program, a pointer may be undefined, associated  
19 with different targets, or be disassociated. If a pointer is associated with a target, the definition status  
20 of the pointer is either defined or undefined, depending on the definition status of the target. If the  
21 pointer has deferred type parameters or shape, their values are assumed from the target. If the pointer  
22 is polymorphic, its dynamic type is the dynamic type of the target.

#### 23 16.5.2.2 Pointer association status

24 A pointer may have a **pointer association status** of associated, disassociated, or undefined. Its  
25 association status may change during execution of a program. Unless a pointer is initialized (explicitly  
26 or by default), it has an initial association status of undefined. A pointer may be initialized to have an  
27 association status of disassociated or associated.

#### NOTE 16.14

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 standard 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.

### 1 16.5.2.2.1 Events that cause pointers to become associated

2 A pointer becomes associated when any of the following events occur.

- 3 • The pointer is allocated (6.3.1) as the result of the successful execution of an ALLOCATE statement  
4 referencing the pointer.
- 5 • The pointer is pointer-assigned to a target (7.4.2) that is associated or is specified with the TAR-  
6 GET attribute and, if allocatable, is allocated.
- 7 • A nonpointer scalar object of type bits with a kind type parameter that is an integer multiple, N,  
8 of the size of a numeric storage unit occupies N contiguous numeric storage units. The ordering of  
9 these consecutive numeric storage units is processor dependent. A nonpointer scalar object of type  
10 bits with a kind type parameter that is not an integer multiple of the size of a numeric storage  
11 unit occupies an unspecified storage unit that is different for each such kind value.

#### J3 internal note

##### Unresolved Technical Issue 75

The “ordering” sentence is trying to say something subtle about endianness.  $X(1)//X(2)$  gives the same result on all machines, but whether when that is stored  $X(1)$  or  $X(2)$  occurs first is processor (endian) dependent.

This should be reworded as the technical effect in the user program, which is maybe only visible as

```

BITS X(2)
BITS(KIND(X)*2) Y,ZLE,ZBE
...
X = TRANSFER(Y,X)
ZLE = X(1)//X(2)
ZBE = X(2)//X(1)
...
! On some processors Y==ZLE, and on other processors Y==ZBE.
! Inconceivably, it might differ from both.
```

- 12 • The pointer is a dummy argument and its corresponding actual argument is not a pointer.
- 13 • The pointer is an ultimate component of an object of a type for which default initialization is  
14 specified for the component, and the corresponding initializer is an initialization target, and
  - 15 – a procedure is invoked with this object as an actual argument corresponding to a nonpointer  
16 nonallocatable dummy argument with INTENT (OUT),
  - 17 – a procedure with this object as an unsaved nonpointer nonallocatable local object that is not  
18 accessed by use or host association is invoked, or
  - 19 – this object is allocated.

#### J3 internal note

##### Unresolved Technical Issue 76

Does defining a bits storage unit associated with a numeric storage unit define the numeric value? Presumably not. What about the other way round? Maybe, but no words to support that.

How does this interact with argument association - since the types are different, defining the dummy isn't going to define the actual!

#### 1 16.5.2.2.2 Events that cause pointers to become disassociated

2 A pointer becomes disassociated when

- 3 (1) the pointer is nullified (6.3.2),
- 4 (2) the pointer is deallocated (6.3.3),
- 5 (3) the pointer is pointer-assigned (7.4.2) to a disassociated pointer, or
- 6 (4) the pointer is an ultimate component of an object of a type for which default initialization is  
7 specified for the component, and the corresponding initializer is a reference to the intrinsic  
8 function NULL, and
  - 9 •a procedure is invoked with this object as an actual argument corresponding to a  
10 nonpointer nonallocatable dummy argument with INTENT (OUT),
  - 11 •a procedure with this object as an unsaved nonpointer nonallocatable local object  
12 that is not accessed by use or host association is invoked, or
  - 13 •this object is allocated.

#### 14 16.5.2.2.3 Events that cause the association status of pointers to become undefined

15 The association status of a pointer becomes undefined when

- 16 (1) the pointer is pointer-assigned to a target that has an undefined association status,
- 17 (2) the target of the pointer is deallocated other than through the pointer,
- 18 (3) the allocation transfer procedure (13.7.124) is executed, the pointer is associated with the  
19 argument FROM, and an object without the target attribute associated with the argument  
20 TO,
- 21 (4) execution of a RETURN or END statement causes the pointer's target to become undefined  
22 (item (3) of 16.6.6),
- 23 (5) execution of the host instance of a procedure pointer is completed by execution of a RE-  
24 TURN or END statement,
- 25 (6) a procedure is terminated by execution of a RETURN or END statement and the pointer  
26 is declared or accessed in the subprogram that defines the procedure unless the pointer
  - 27 (a) has the SAVE attribute,
  - 28 (b) is in blank common,
  - 29 (c) is in a named common block that appears in at least one other scoping unit that is  
30 in execution,
  - 31 (d) is in the scoping unit of a module if the module also is accessed by another scoping  
32 unit that is in execution,
  - 33 (e) is in the scoping unit of a submodule if any scoping unit in that submodule or any of  
34 its descendant submodules is in execution,
  - 35 (f) is accessed by host association, or
  - 36 (g) is the return value of a function declared to have the POINTER attribute,
- 37 (7) the pointer is an ultimate component of an object, default initialization is not specified  
38 for the component, and a procedure is invoked with this object as an actual argument  
39 corresponding to a dummy argument with INTENT(OUT), or
- 40 (8) a procedure is invoked with the pointer as an actual argument corresponding to a pointer  
41 dummy argument with INTENT(OUT).

#### 42 16.5.2.2.4 Other events that change the association status of pointers

43 When a pointer becomes associated with another pointer by argument association, construct association,  
44 or host association, the effects on its association status are specified in 16.5.5.

1 While two pointers are name associated, storage associated, or inheritance associated, if the association  
 2 status of one pointer changes, the association status of the other changes accordingly.

### 3 16.5.2.3 Pointer definition status

4 The definition status of a pointer is that of its target. If a pointer is associated with a definable target,  
 5 the definition status of the pointer may be defined or undefined according to the rules for a variable  
 6 (16.6).

### 7 16.5.2.4 Relationship between association status and definition status

8 If the association status of a pointer is disassociated or undefined, the pointer shall not be referenced  
 9 or deallocated. Whatever its association status, a pointer always may be nullified, allocated, or pointer  
 10 assigned. A nullified pointer is disassociated. When a pointer is allocated, it becomes associated but  
 11 undefined. When a pointer is pointer assigned, its association and definition status become those of the  
 12 specified *data-target* or *proc-target*.

## 13 16.5.3 Storage association

### 14 16.5.3.1 General

15 Storage sequences are used to describe relationships that exist among variables, common blocks, and  
 16 result variables. **Storage association** is the association of two or more data objects that occurs when  
 17 two or more storage sequences share or are aligned with one or more storage units.

### 18 16.5.3.2 Storage sequence

19 A **storage sequence** is a sequence of storage units. The **size of a storage sequence** is the number  
 20 of storage units in the storage sequence. A **storage unit** is a character storage unit, a numeric storage  
 21 unit, a file storage unit(9.2.4), or an unspecified storage unit. The sizes of the numeric storage unit, the  
 22 character storage unit and the file storage unit are the value of constants in the ISO\_FORTRAN\_ENV  
 23 intrinsic module (13.8.3).

24 In a storage association context

- 25 (1) a nonpointer scalar object of type default integer, default real, default logical, or default  
 26 bits occupies a single **numeric storage unit**,
- 27 (2) a nonpointer scalar object of type double precision real or default complex occupies two  
 28 contiguous numeric storage units,
- 29 (3) a nonpointer scalar object of type default character and character length *len* occupies *len*  
 30 contiguous **character storage units**,
- 31 (4) a nonpointer scalar object of type character with the C character kind (15.2.2) and character  
 32 length *len* occupies *len* contiguous **unspecified storage units**,
- 33 (5) a nonpointer scalar object of sequence type with no type parameters occupies a sequence of  
 34 storage sequences corresponding to the sequence of its ultimate components,
- 35 (6) a nonpointer scalar object of any type not specified in items (1)-(6) occupies a single un-  
 36 specified storage unit that is different for each case and each set of type parameter values,  
 37 and that is different from the unspecified storage units of item (4),
- 38 (7) a nonpointer array occupies a sequence of contiguous storage sequences, one for each array  
 39 element, in array element order (6.2.2.2), and
- 40 (8) a pointer occupies a single unspecified storage unit that is different from that of any non-  
 41 pointer object and is different for each combination of type, type parameters, and rank. A

- 1 pointer that has the CONTIGUOUS attribute occupies a storage unit that is different from  
 2 that of a pointer that does not have the CONTIGUOUS attribute.
- 3 A sequence of storage sequences forms a storage sequence. The order of the storage units in such a  
 4 composite storage sequence is that of the individual storage units in each of the constituent storage  
 5 sequences taken in succession, ignoring any zero-sized constituent sequences.
- 6 Each common block has a storage sequence (5.7.2.2).
- 7 Two objects for which the intrinsic function BITS\_KIND returns the same value occupy the same amount  
 8 of storage.

**J3 internal note****Unresolved Technical Issue 77**

The purpose of the above paragraph here seems to be to promote rampant nonportability. We have always been at liberty to allow this in the past, and we have never done so, because it runs directly counter to our portability goal.

This effect is to *allow* storage association of non-default kinds of integer, real et al. The fact that BITS\_KIND returns different values on different systems is no consolation to the user who is faced with mysterious failures; even with accurate diagnostics he will face the daunting task of unravelling someone else's nonportable code (and he might not be in a position to unravel it all anyway).

**NOTE 16.15**

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.

**9 16.5.3.3 Association of storage sequences**

- 10 Two nonzero-sized storage sequences  $s_1$  and  $s_2$  are **storage associated** if the  $i$ th storage unit of  $s_1$  is  
 11 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  
 12 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  $1 \leq j + k \leq$   
 13  $\text{size of } s_2$ .
- 14 Storage association also is defined between two zero-sized storage sequences, and between a zero-sized  
 15 storage sequence and a storage unit. A zero-sized storage sequence in a sequence of storage sequences is  
 16 storage associated with its successor, if any. If the successor is another zero-sized storage sequence, the  
 17 two sequences are storage associated. If the successor is a nonzero-sized storage sequence, the zero-sized  
 18 sequence is storage associated with the first storage unit of the successor. Two storage units that are  
 19 each storage associated with the same zero-sized storage sequence are the same storage unit.

**NOTE 16.16**

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
...
```

**NOTE 16.16 (cont.)**

```
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.

**1 16.5.3.4 Association of scalar data objects**

2 Two scalar data objects are storage associated if their storage sequences are storage associated. Two  
3 scalar entities are **totally associated** if they have the same storage sequence. Two scalar entities are  
4 **partially associated** if they are associated without being totally associated.

5 The definition status and value of a data object affects the definition status and value of any storage  
6 associated entity. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement  
7 may cause storage association of storage sequences.

8 An EQUIVALENCE statement causes storage association of data objects only within one scoping unit,  
9 unless one of the equivalenced entities is also in a common block (5.7.1.2 and 5.7.2.2).

10 COMMON statements cause data objects in one scoping unit to become storage associated with data  
11 objects in another scoping unit.

12 A common block is permitted to contain a sequence of differing storage units. All scoping units that  
13 access named common blocks with the same name shall specify an identical sequence of storage units.  
14 Blank common blocks may be declared with differing sizes in different scoping units. For any two blank  
15 common blocks, the initial sequence of storage units of the longer blank common block shall be identical  
16 to the sequence of storage units of the shorter common block. If two blank common blocks are the same  
17 length, they shall have the same sequence of storage units.

18 An ENTRY statement in a function subprogram causes storage association of the result variables.

19 Partial association shall exist only between

- 20 (1) an object of default character or character sequence type and an object of default character  
21 or character sequence type, or
- 22 (2) an object of default complex, double precision real, or numeric sequence type and an object  
23 of default integer, default real, default logical, double precision real, default complex, or  
24 numeric sequence type.

25 For noncharacter entities, partial association may occur only through the use of COMMON, EQUIV-  
26 ALENCE, or ENTRY statements. For character entities, partial association may occur only through  
27 argument association or the use of COMMON or EQUIVALENCE statements.

**NOTE 16.17**

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

**NOTE 16.17 (cont.)**

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.

- 1 Partial association of character entities occurs when some, but not all, of the storage units of the entities
- 2 are the same.

**NOTE 16.18**

In the example:

```
CHARACTER A*4, B*4, C*3
EQUIVALENCE (A (2:3), B, C)
```

A, B, and C are partially associated.

- 3 A storage unit shall not be explicitly initialized more than once in a program. Explicit initialization
- 4 overrides default initialization, and default initialization for an object of derived type overrides default
- 5 initialization for a component of the object (4.5.2). Default initialization may be specified for a storage
- 6 unit that is storage associated provided the objects supplying the default initialization are of the same
- 7 type and type parameters, and supply the same value for the storage unit.

**8 16.5.4 Inheritance association**

- 9 Inheritance association occurs between components of the parent component and components inherited
- 10 by type extension into an extended type (4.5.7.2). This association is persistent; it is not affected by the
- 11 accessibility of the inherited components.

**12 16.5.5 Establishing associations**

- 13 When an association is established between two entities by argument association, host association,
- 14 or construct association, certain characteristics of the **associating entity** become those of the **pre-**
- 15 **existing** entity.

- 16 For argument association, if the dummy argument is not a pointer and the corresponding actual argument
- 17 is a pointer, the pre-existing entity is the target of that pointer. Otherwise the pre-existing entity is the
- 18 corresponding actual argument. In either case, the associating entity is the dummy argument.

- 19 For host association, the associating entity is the entity in the host scoping unit and the pre-existing
- 20 entity is the entity in the contained scoping unit. If an internal procedure is invoked via a dummy
- 21 procedure or procedure pointer, the pre-existing entity that participates in the association is the one
- 22 from the host instance. Otherwise, if the host scoping unit is a recursive procedure, the pre-existing entity



1 that participates in the association is the one from the innermost subprogram instance that invoked,  
2 directly or indirectly, the contained procedure.

3 For construct association, the associating entity is identified by the associate name and the pre-existing  
4 entity is the selector.

5 When an association is established by argument association, host association, or construct association,  
6 the following applies.

7       (1) If the associating entity has the `POINTER` attribute, its pointer association status becomes  
8       the same as that of the pre-existing entity. If the pre-existing entity has a pointer association  
9       status of associated, the associating entity becomes pointer associated with the same target  
10      and, if they are arrays, the bounds of the associating entity become the same as those of  
11      the pre-existing entity.

12      (2) If the associating entity has the `ALLOCATABLE` attribute, its allocation status becomes  
13      the same as that of the pre-existing entity. If the pre-existing entity is allocated, the bounds  
14      (if it is an array), values of deferred type parameters, definition status, and value (if it is  
15      defined) become the same as those of the pre-existing entity. If the associating entity is  
16      polymorphic and the pre-existing entity is allocated, the dynamic type of the associating  
17      entity becomes the same as that of the pre-existing entity.

18 If the associating entity is neither a pointer nor allocatable, its definition status and value (if it is defined)  
19 become the same as those of the pre-existing entity. If the entities are arrays and the association is not  
20 argument association, the bounds of the associating entity become the same as those of the pre-existing  
21 entity.

## 22 **16.6 Definition and undefinition of variables**

### 23 **16.6.1 Definition of objects and subobjects**

24 A variable may be defined or may be undefined and its definition status may change during execution of  
25 a program. An action that causes a variable to become undefined does not imply that the variable was  
26 previously defined. An action that causes a variable to become defined does not imply that the variable  
27 was previously undefined.

28 Arrays, including sections, and variables of derived, character, or complex type are objects that consist of  
29 zero or more subobjects. Associations may be established between variables and subobjects and between  
30 subobjects of different variables. These subobjects may become defined or undefined.

31 An array is defined if and only if all of its elements are defined.

32 A derived-type scalar object is defined if and only if all of its nonpointer components are defined.

33 A complex or character scalar object is defined if and only if all of its subobjects are defined.

34 If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

### 35 **16.6.2 Variables that are always defined**

36 Zero-sized arrays and zero-length strings are always defined.

### 37 **16.6.3 Variables that are initially defined**

38 The following variables are initially defined:

- 1 (1) variables specified to have initial values by DATA statements;
- 2 (2) variables specified to have initial values by type declaration statements;
- 3 (3) nonpointer default-initialized subcomponents of variables that do not have the ALLOCAT-
- 4 ABLE or POINTER attribute, and are saved, local variables of a main program, or declared
- 5 in a MODULE or BLOCK DATA scoping unit;
- 6 (4) pointers specified to be initially associated with a variable that is initially defined;
- 7 (5) variables that are always defined;
- 8 (6) variables with the BIND attribute that are initialized by means other than Fortran.

**NOTE 16.19**

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.

## 9 16.6.4 Variables that are initially undefined

- 10 All other variables are initially undefined.

## 11 16.6.5 Events that cause variables to become defined

- 12 Variables become defined by the following events.

- 13 (1) Execution of an intrinsic assignment statement other than a masked array assignment or
- 14 FORALL assignment statement causes the variable that precedes the equals to become
- 15 defined. Execution of a defined assignment statement may cause all or part of the variable
- 16 that precedes the equals to become defined.
- 17 (2) Execution of a masked array assignment or FORALL assignment statement may cause some
- 18 or all of the array elements in the assignment statement to become defined (7.4.3).
- 19 (3) As execution of an input statement proceeds, each variable that is assigned a value from
- 20 the input file becomes defined at the time that data is transferred to it. (See (4) in 16.6.6.)
- 21 Execution of a WRITE statement whose unit specifier identifies an internal file causes each
- 22 record that is written to become defined.
- 23 (4) Execution of a DO statement causes the DO variable, if any, to become defined.
- 24 (5) Beginning of execution of the action specified by an *io-implied-do* in a synchronous in-
- 25 put/output statement causes the *do-variable* to become defined.
- 26 (6) A reference to a procedure causes the entire dummy argument data object to become defined
- 27 if the dummy argument does not have INTENT(OUT) and the entire corresponding actual
- 28 argument is defined.
- 29 A reference to a procedure causes a subobject of a dummy argument to become defined if
- 30 the dummy argument does not have INTENT(OUT) and the corresponding subobject of
- 31 the corresponding actual argument is defined.

- 1       (7)   Execution of an input/output statement containing an IOSTAT= specifier causes the spec-
- 2       specified integer variable to become defined.
- 3       (8)   Execution of a synchronous READ statement containing a SIZE= specifier causes the spec-
- 4       specified integer variable to become defined.
- 5       (9)   Execution of a wait operation corresponding to an asynchronous input statement containing
- 6       a SIZE= specifier causes the specified integer variable to become defined.
- 7       (10)   Execution of an INQUIRE statement causes any variable that is assigned a value during the
- 8       execution of the statement to become defined if no error condition exists.
- 9       (11)   If an error, end-of-file, or end-of-record condition occurs during execution of an input/output
- 10      statement that has an IOMSG= specifier, the *iomsg-variable* becomes defined.
- 11      (12)   When a character storage unit becomes defined, all associated character storage units be-
- 12      come defined.
- 13      When a numeric storage unit becomes defined, all associated numeric storage units of the
- 14      same type become defined. When an entity of double precision real type becomes defined,
- 15      all totally associated entities of double precision real type become defined.
- 16      When an unspecified storage unit becomes defined, all associated unspecified storage units
- 17      become defined.
- 18      (13)   When a default complex entity becomes defined, all partially associated default real entities
- 19      become defined.
- 20      (14)   When both parts of a default complex entity become defined as a result of partially associ-
- 21      ated default real or default complex entities becoming defined, the default complex entity
- 22      becomes defined.
- 23      (15)   When all components of a structure of a numeric sequence type or character sequence type
- 24      become defined as a result of partially associated objects becoming defined, the structure
- 25      becomes defined.
- 26      (16)   Execution of an ALLOCATE, DEALLOCATE, NOTIFY, QUERY, SYNC\_ALL, SYNC\_IMAGES,
- 27      SYNC\_MEMORY, or SYNC\_TEAM statement with a STAT= specifier causes the variable
- 28      specified by the STAT= specifier to become defined.
- 29      (17)   If an error condition occurs during execution of an ALLOCATE, DEALLOCATE, NOTIFY,
- 30      QUERY, SYNC\_ALL, SYNC\_IMAGES, SYNC\_MEMORY, or SYNC\_TEAM statement that
- 31      has an ERRMSG= specifier, the variable specified by the ERRMSG= specifier becomes
- 32      defined.
- 33      (18)   Allocation of a zero-sized array causes the array to become defined.
- 34      (19)   Allocation of an object that has a nonpointer default-initialized subcomponent causes that
- 35      subcomponent to become defined.
- 36      (20)   Invocation of a procedure causes any automatic object of zero size in that procedure to
- 37      become defined.
- 38      (21)   Execution of a pointer assignment statement that associates a pointer with a target that is
- 39      defined causes the pointer to become defined.
- 40      (22)   Invocation of a procedure that contains an unsaved nonpointer nonallocatable local variable
- 41      causes all nonpointer default-initialized subcomponents of the object to become defined.
- 42      (23)   Invocation of a procedure that has a nonpointer nonallocatable INTENT (OUT) dummy
- 43      argument causes all nonpointer default-initialized subcomponents of the dummy argument
- 44      to become defined.
- 45      (24)   Invocation of a nonpointer function of a derived type causes all nonpointer default-initialized
- 46      subcomponents of the function result to become defined.
- 47      (25)   In a FORALL construct, the *index-name* becomes defined when the *index-name* value set
- 48      is evaluated.
- 49      (26)   An object with the VOLATILE attribute that is changed by a means not specified by the
- 50      program becomes defined (see 5.3.18).

- 1 (27) Execution of the BLOCK statement of a BLOCK construct that has an unsaved nonpointer  
2 nonallocatable local variable causes all nonpointer default-initialized subcomponents of the  
3 variable to become defined.
- 4 (28) Execution of an OPEN statement containing a NEWUNIT= specifier causes the specified  
5 integer variable to become defined.

## 6 16.6.6 Events that cause variables to become undefined

7 Variables become undefined by the following events.

- 8 (1) When a variable of a given type becomes defined, all associated variables of different type  
9 become undefined. However, when a variable of type default real is partially associated with  
10 a variable of type default complex, the complex variable does not become undefined when  
11 the real variable becomes defined and the real variable does not become undefined when  
12 the complex variable becomes defined. When a variable of type default complex is partially  
13 associated with another variable of type default complex, definition of one does not cause  
14 the other to become undefined.
- 15 (2) If the evaluation of a function may cause a variable to become defined and if a reference to  
16 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,  
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,  
24 (c) unsaved nonfinalizable local variables of a module or submodule become undefined  
25 unless another active scoping unit is referencing the module or submodule, and

### NOTE 16.20

A module subprogram inherently references the module that is its host. Therefore, for processors that keep track of when modules are in use, a module is in use whenever any procedure in the module is active, even if no other active scoping units reference the module; this situation can arise if a module procedure is invoked via a procedure pointer, a type-bound procedure, or a companion processor.

- 26 (d) unsaved finalizable local variables of a module or submodule may be finalized if no  
27 other active scoping unit is referencing the module or submodule – following which  
28 they become undefined.
- 29 (4) When an error condition or end-of-file condition occurs during execution of an input state-  
30 ment, all of the variables specified by the input list or namelist group of the statement  
31 become undefined.
- 32 (5) When an error condition, end-of-file condition, or end-of-record condition occurs during  
33 execution of an input/output statement and the statement contains any *io-implied-dos*, all  
34 of the *do-variables* in the statement become undefined (9.10).
- 35 (6) Execution of a direct access input statement that specifies a record that has not been written  
36 previously causes all of the variables specified by the input list of the statement to become  
37 undefined.
- 38 (7) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC=  
39 variables to become undefined (9.9).
- 40 (8) When a character storage unit becomes undefined, all associated character storage units  
41 become undefined.

- 1           When a numeric storage unit becomes undefined, all associated numeric storage units be-  
2           come undefined unless the undefinition is a result of defining an associated numeric storage  
3           unit of different type (see (1) above).
- 4           When an entity of double precision real type becomes undefined, all totally associated  
5           entities of double precision real type become undefined.
- 6           When an unspecified storage unit becomes undefined, all associated unspecified storage units  
7           become undefined.
- 8       (9)   When an allocatable entity is deallocated, it becomes undefined.
- 9       (10) When the allocation transfer procedure (13.5.17) causes the allocation status of an allocat-  
10       able entity to become unallocated, the entity becomes undefined.
- 11       (11) Successful execution of an ALLOCATE statement for a nonzero-sized object that has a sub-  
12       component for which default initialization has not been specified causes the subcomponent  
13       to become undefined.
- 14       (12) Execution of an INQUIRE statement causes all inquiry specifier variables to become un-  
15       defined if an error condition exists, except for any variable in an IOSTAT= or IOMSG=  
16       specifier.
- 17       (13) When a procedure is invoked
- 18           (a)   an optional dummy argument that is not associated with an actual argument becomes  
19           undefined,
- 20           (b)   a dummy argument with INTENT (OUT) becomes undefined except for any non-  
21           pointer default-initialized subcomponents of the argument,
- 22           (c)   an actual argument associated with a dummy argument with INTENT (OUT) be-  
23           comes undefined except for any nonpointer default-initialized subcomponents of the  
24           argument,
- 25           (d)   a subobject of a dummy argument that does not have INTENT (OUT) becomes  
26           undefined if the corresponding subobject of the actual argument is undefined, and
- 27           (e)   the result variable of a function becomes undefined except for any of its nonpointer  
28           default-initialized subcomponents.
- 29       (14) When the association status of a pointer becomes undefined or disassociated (16.5.2.2.2-  
30       16.5.2.2.3), the pointer becomes undefined.
- 31       (15) When the execution of a FORALL or DO CONCURRENT construct completes, the *index-*  
32       *names* become undefined.
- 33       (16) When a DO CONCURRENT construct terminates, a variable that is defined or becomes  
34       undefined during more than one iteration of the construct becomes undefined.
- 35       (17) Execution of an asynchronous READ statement causes all of the variables specified by the  
36       input list or SIZE= specifier to become undefined. Execution of an asynchronous namelist  
37       READ statement causes any variable in the namelist group to become undefined if that  
38       variable will subsequently be defined during the execution of the READ statement or the  
39       corresponding WAIT operation.
- 40       (18) When execution of a RETURN or END statement causes a variable to become undefined,  
41       any variable of type C\_PTR becomes undefined if its value is the C address of any part of  
42       the variable that becomes undefined.
- 43       (19) When a variable with the TARGET attribute is deallocated, any variable of type C\_PTR  
44       becomes undefined if its value is the C address of any part of the variable that is deallocated.
- 45       (20) When a BLOCK construct terminates, its unsaved local variables become undefined.

**NOTE 16.21**

Execution of a defined assignment statement may leave all or part of the variable undefined.

## 1 16.6.7 Variable definition context

2 Some variables are prohibited from appearing in a syntactic context that would imply definition or  
 3 undefined of the variable (5.3.9, 5.3.14, 12.7). The following are the contexts in which the appearance  
 4 of a variable implies such definition or undefined of the variable:

- 5 • the *variable* of an *assignment-stmt*;
- 6 • a *pointer-object* in a *nullify-stmt*;
- 7 • a *data-pointer-object* or *proc-pointer-object* in a *pointer-assignment-stmt*;
- 8 • a *do-variable* in a *do-stmt* or *io-implied-do*;
- 9 • an *input-item* in a *read-stmt*;
- 10 • a *variable-name* in a *namelist-stmt* if the *namelist-group-name* appears in a NML= specifier in a  
 11 *read-stmt*;
- 12 • an *internal-file-variable* in a *write-stmt*;
- 13 • an IOSTAT=, SIZE=, or IOMSG= specifier in an input/output statement;
- 14 • a specifier in an INQUIRE statement other than FILE=, ID=, and UNIT=;
- 15 • a NEWUNIT= specifier in an OPEN statement;
- 16 • a READY= specifier in a QUERY statement;
- 17 • a *stat-variable*, *allocate-object*, or *errmsg-variable*;
- 18 • an actual argument in a reference to a procedure with an explicit interface if the associated dummy  
 19 argument has the INTENT(OUT) or INTENT(INOUT) attribute;
- 20 • a *variable* that is the *selector* in a SELECT TYPE or ASSOCIATE construct if the associate name  
 21 of that construct appears in a variable definition context.

## 22 16.6.8 Pointer association context

23 Some pointers are prohibited from appearing in a syntactic context that would imply alteration of the  
 24 pointer association status (16.5.2.2, 5.3.9, 5.3.14). The following are the contexts in which the appearance  
 25 of a pointer implies such alteration of its pointer association status:

- 26 • a *pointer-object* in a *nullify-stmt*;
- 27 • a *data-pointer-object* or *proc-pointer-object* in a *pointer-assignment-stmt*;
- 28 • an *allocate-object* in an *allocate-stmt* or *deallocate-stmt*;
- 29 • an actual argument in a reference to a procedure if the associated dummy argument is a pointer  
 30 with the INTENT (OUT) or INTENT (INOUT) attribute.

31 If a reference to a function appears in a variable definition context the result of the function reference  
 32 shall be a pointer that is associated with a definable target. That target is the variable that becomes  
 33 defined or undefined.

## Annex A

(Informative)

### Glossary of technical terms

The following is a list of the principal technical terms used in the standard 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 in the standard.

**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.

**action statement** (2.1) : A single statement specifying or controlling a computational action (R214).

**actual argument** (12, 12.5.1) : 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, 6.2.2.3) : A subobject that is an array and is not a structure component.

**assignment statement** (7.4.1.1) : A statement that evaluates an expression and assigns its value to a variable.

**associate name** (8.1.5.1) : 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.

- 1 **attribute** (5) : A property of an entity that determines its uses.
- 2 **automatic data object** (5.2) : A data object that is a local entity of a subprogram, that is not a dummy  
3 argument, and that has a length type parameter or array bound that is specified by an expression that  
4 is not an initialization expression.
- 5 **base type** (4.5.7) : An extensible type that is not an extension of another type.
- 6 **belong** (8.1.7.5.3, 8.1.7.5.4) : If an EXIT or a CYCLE statement contains a construct name, the  
7 statement **belongs** to the construct using that name. Otherwise, it **belongs** to the innermost DO  
8 construct in which it appears.
- 9 **binding label** (15.5.2, 15.4.2) : A value of type default character that uniquely identifies how a variable,  
10 common block, subroutine, or function is known to a companion processor.
- 11 **block** (8.1) : A sequence of executable constructs embedded in another executable construct, bounded  
12 by statements that are particular to the construct, and treated as an integral unit.
- 13 **block data program unit** (11.3) : A program unit that provides initial values for data objects in  
14 named common blocks.
- 15 **bounds** (5.3.7.2) : For a named array, the limits within which the values of the subscripts of its array  
16 elements shall lie.
- 17 **character** (3.1) : A letter, digit, or other symbol.
- 18 **character length parameter** (2.4.1.1) : The type parameter that specifies the number of characters  
19 for an entity of type character.
- 20 **character storage unit** (16.5.3.2) : The unit of storage for holding a scalar that is not a pointer and  
21 is of type default character and character length one.
- 22 **character string** (4.4.5) : A sequence of characters numbered from left to right 1, 2, 3, ...
- 23 **characteristics** (12.3) :
- 24 Of a procedure, its classification as a function or subroutine, whether it is pure, whether it is elemental,  
25 whether it has the BIND attribute, the value of its binding label, the characteristics of its dummy  
26 arguments, and the characteristics of its function result if it is a function.
- 27 Of a dummy argument, whether it is a data object, is a procedure, is a procedure pointer, is an asterisk  
28 (alternate return indicator), or has the OPTIONAL attribute.
- 29 Of a dummy data object, its type, type parameters, shape, the exact dependence of an array bound  
30 or type parameter on other entities, intent, whether it is optional, whether it is a pointer or a target,  
31 whether it is allocatable, whether it has the VALUE, ASYNCHRONOUS, or VOLATILE attributes,  
32 whether it is polymorphic, and whether the shape, size, or a type parameter is assumed.
- 33 Of a dummy procedure or procedure pointer, whether the interface is explicit, the characteristics of the  
34 procedure if the interface is explicit, and whether it is optional.
- 35 Of a function result, its type, type parameters, which type parameters are deferred, whether it is poly-  
36 morphic, whether it is a pointer or allocatable, whether it is a procedure pointer, rank if it is a pointer  
37 or allocatable, shape if it is not a pointer or allocatable, the exact dependence of an array bound or type  
38 parameter on other entities, and whether the character length is assumed.
- 39 **class** (4.3.1.3) : A class named N is the set of types extended from the type named N.
- 40 **co-array** (2.4.6) A data entity that has nonzero co-rank. A non-dummy co-array has the same shape



- 1 on every image and its values may be referenced or defined by any image.
- 2 **co-bounds** (5.3.7) : For a co-array, the limits within which the values of the co-subscripts of its co-  
3 indexed objects shall lie.
- 4 **co-indexed object** (2.4.6) : An object whose designator includes an *image-selector*.
- 5 **co-rank** (2.4.6) : The number of co-dimensions of a co-array.
- 6 **co-subscript** (6.2.3) : One of the list of scalar integer expressions in an *image-selector*.
- 7 **collating sequence** (4.4.5.4) : An ordering of all the different characters of a particular kind type  
8 parameter.
- 9 **collective subroutine** (13.1) : An intrinsic subroutine that has an argument of type IMAGE\_TEAM.
- 10 **common block** (5.7.2) : A block of physical storage that may be accessed by any of the scoping units  
11 in a program.
- 12 **companion processor** (2.5.10): A mechanism by which global data and procedures may be referenced  
13 or defined. It may be a mechanism that references and defines such entities by means other than Fortran.  
14 The procedures can be described by a C function prototype.
- 15 **component** (4.5) : A constituent of a derived type.
- 16 **component order** (4.5.4.6) : The ordering of the components of a derived type that is used for intrinsic  
17 formatted input/output and for structure constructors.
- 18 **conformable** (2.4.5) : Two arrays are said to be **conformable** if they have the same shape. A scalar  
19 is conformable with any array.
- 20 **conformance** (1.5) : A program conforms to the standard if it uses only those forms and relationships  
21 described therein and if the program has an interpretation according to the standard. A program unit  
22 conforms to the standard if it can be included in a program in a manner that allows the program to be  
23 standard-conforming. A processor conforms to the standard if it executes standard-conforming programs  
24 in a manner that fulfills the interpretations prescribed in the standard and contains the capability of  
25 detection and reporting as listed in 1.5.
- 26 **connect team** (9.4.6.7) : The set of images that are permitted to reference a particular external  
27 input/output unit.
- 28 **connected** (9.4.3) :
- 29 For an external unit, the property of referring to an external file.
- 30 For an external file, the property of having an external unit that refers to it.
- 31 **constant** (2.4.3.1.2) : A data object whose value shall not change during execution of a program. It  
32 may be a named constant or a literal constant.
- 33 **construct** (7.4.3, 7.4.4, 8.1) : A sequence of statements starting with an ASSOCIATE, BLOCK, DO,  
34 FORALL, IF, SELECT CASE, SELECT TYPE, or WHERE statement and ending with the correspond-  
35 ing terminal statement.
- 36 **construct association** (16.5.1.6) : The association between the selector of an ASSOCIATE or SELECT  
37 TYPE construct and the associate name.
- 38 **construct entity** (16) : An entity defined by a lexical token whose scope is a construct.

- 1 **control mask** (7.4.3) : In a WHERE statement or construct, an array of type logical whose value  
2 determines which elements of an array, in a *where-assignment-stmt*, will be defined.
- 3 **data** : Plural of datum.
- 4 **data entity** (2.4.3) : A data object, the result of the evaluation of an expression, or the result of the  
5 execution of a function reference (called the function result). A data entity has a type (either intrinsic  
6 or derived) and has, or may have, a data value (the exception is an undefined variable). Every data  
7 entity has a rank and is thus either a scalar or an array.
- 8 **data object** (2.4.3.1) : A data entity that is a constant, a variable, or a subobject of a constant.
- 9 **data type** (4) : See type.
- 10 **datum** : A single quantity that may have any of the set of values specified for its type.
- 11 **decimal symbol** (9.9.1.6, 10.6, 10.8.8) : The character that separates the whole and fractional parts in  
12 the decimal representation of a real number in a file. By default the decimal symbol is a decimal point  
13 (also known as a period). The current decimal symbol is determined by the current decimal edit mode.
- 14 **declared type** (4.3.1.3, 7.1.4) : The type that a data entity is declared to have. May differ from the  
15 type during execution (the dynamic type) for polymorphic data entities.
- 16 **default initialization** (4.5) : If initialization is specified in a type definition, an object of the type will  
17 be automatically initialized. Nonpointer components may be initialized with values by default; pointer  
18 components may be initially disassociated by default. Default initialization is not provided for objects  
19 of intrinsic type.
- 20 **default-initialized** (4.5.4.5) : A subcomponent is said to be default-initialized if it will be initialized  
21 by default initialization.
- 22 **deferred binding** (4.5.5) : A binding with the DEFERRED attribute. A deferred binding shall appear  
23 only in an abstract type definition (4.5.7).
- 24 **deferred type parameter** (4.3) : A length type parameter whose value is not specified in the decla-  
25 ration of an object, but instead is specified when the object is allocated or pointer-assigned.
- 26 **definable** (2.5.5) : A variable is **definable** if its value may be changed by the appearance of its  
27 designator on the left of an assignment statement. An allocatable variable that has not been allocated  
28 is an example of a data object that is not definable. An example of a subobject that is not definable is  
29 C (I) when C is an array that is a constant and I is an integer variable.
- 30 **defined** (2.5.5) : For a data object, the property of having or being given a valid value.
- 31 **defined assignment statement** (7.4.1.4, 12.4.2.1.2) : An assignment statement that is not an intrinsic  
32 assignment statement; it is defined by a subroutine and a generic interface that specifies ASSIGNMENT  
33 (=).
- 34 **defined operation** (7.1.3, 12.4.2.1.1) : An operation that is not an intrinsic operation and is defined  
35 by a function that is associated with a generic identifier.
- 36 **deleted feature** (1.8) : A feature in a previous Fortran standard that is considered to have been  
37 redundant and largely unused. See B.1 for a list of features that are in a previous Fortran standard, but  
38 are not in this standard. A feature designated as an obsolescent feature in the standard may become a  
39 deleted feature in the next revision.
- 40 **derived type** (2.4.1.2, 4.5) : A type whose data have components, each of which is either of intrinsic  
41 type or of another derived type.

- 1 **designator** (2.5.1) : A name, followed by zero or more subobject selectors.
- 2 **disassociated** (2.4.7) : A disassociated pointer is not associated with a target. A pointer is disassociated  
3 following execution of a NULLIFY statement, following pointer assignment with a disassociated pointer,  
4 by default initialization, or by explicit initialization. A data pointer may also be disassociated by  
5 execution of a DEALLOCATE statement.
- 6 **dummy argument** (12, 12.6.2.1, 12.6.2.2, 12.6.2.5, 12.6.4) : An entity by which an associated actual  
7 argument is accessed during execution of a procedure.
- 8 **dummy array** : A dummy argument that is an array.
- 9 **dummy data object** (12.3.1.1, 12.5.1.4) : A dummy argument that is a data object.
- 10 **dummy procedure** (12.2.2.3) : A dummy argument that is specified or referenced as a procedure.
- 11 **dynamic type** (4.3.1.3, 7.1.4) : The type of a data entity during execution of a program. The dynamic  
12 type of a data entity that is not polymorphic is the same as its declared type.
- 13 **effective item** (9.5.2) : A scalar object resulting from expanding an input/output list according to the  
14 rules in 9.5.2.
- 15 **elemental** (2.4.5, 7.4.1.4, 12.8) : An adjective applied to an operation, procedure, or assignment state-  
16 ment that is applied independently to elements of an array or corresponding elements of a set of con-  
17 formable arrays and scalars.
- 18 **entity** : The term used for any of the following: an abstract interface, common block, construct,  
19 data entity, external unit, generic interface,, macro namelist group, operator, procedure, program unit,  
20 statement function, statement label, or type.
- 21 **executable construct** (2.1) : An action statement (R214) or an ASSOCIATE, CASE, DO, FORALL,  
22 IF, SELECT TYPE, or WHERE construct.
- 23 **executable statement** (2.3.1) : An instruction to perform or control one or more computational  
24 actions.
- 25 **explicit initialization** (5.2) : Explicit initialization may be specified for objects of intrinsic or derived  
26 type in type declaration statements or DATA statements. An object of a derived type that specifies  
27 default initialization shall not appear in a DATA statement.
- 28 **explicit interface** (12.4.1) : If a procedure has an explicit interface at the point of a reference to it, the  
29 processor is able to verify that the characteristics of the reference and declaration are related as required  
30 by this standard. A procedure has an explicit interface if it is an internal procedure, a module procedure,  
31 an intrinsic procedure, an external procedure that has an interface body, a procedure reference in its  
32 own scoping unit, or a dummy procedure that has an interface body.
- 33 **explicit-shape array** (5.3.7.2) : A named array that is declared with explicit bounds.
- 34 **expression** (2.4.3.2, 7.1) : A sequence of operands, operators, and parentheses (R722). It may be a  
35 variable, a constant, a function reference, or may represent a computation.
- 36 **extended type** (4.5.7) : An extensible type that is an extension of another type. A type that is declared  
37 with the EXTENDS attribute.
- 38 **extensible type** (4.5.7) : A type from which new types may be derived using the EXTENDS attribute.  
39 A nonsequence type that does not have the BIND attribute.
- 40 **extension type** (4.5.7) : A base type is an extension type of itself only. An extended type is an

- 1 extension type of itself and of all types for which its parent type is an extension.
- 2 **extent** (2.4.5) : The size of one dimension of an array.
- 3 **external file** (9.2) : A sequence of records that exists in a medium external to the program.
- 4 **external linkage** : The characteristic describing that a C entity is global to the program; defined in  
5 subclause 6.2.2 of the C International Standard.
- 6 **external procedure** (2.2.3.1) : A procedure that is defined by an external subprogram or by a means  
7 other than Fortran.
- 8 **external subprogram** (2.2) : A subprogram that is not in a main program, module, or another  
9 subprogram. Note that a module is not called a subprogram. Note that in FORTRAN 77, a block data  
10 program unit is called a subprogram.
- 11 **external unit** (9.4) : A mechanism that is used to refer to an external file. It is identified by a  
12 nonnegative integer.
- 13 **file** (9) : An internal file or an external file.
- 14 **file storage unit** (9.2.4) : The unit of storage for an unformatted or stream file.
- 15 **final subroutine** (4.5.6) : A subroutine that is called automatically by the processor during finalization.
- 16 **finalizable** (4.5.6) : A type that has final subroutines, or that has a finalizable component. An object  
17 of finalizable type.
- 18 **finalization** (4.5.6.2) : The process of calling user-defined final subroutines immediately before destroy-  
19 ing an object.
- 20 **function** (2.2.3) : A procedure that is invoked in an expression and computes a value which is then  
21 used in evaluating the expression.
- 22 **function result** (12.6.2.1) : The data object that returns the value of a function.
- 23 **function subprogram** (12.6.2.1) : A sequence of statements beginning with a FUNCTION statement  
24 that is not in an interface block and ending with the corresponding END statement.
- 25 **generic identifier** (12.4.2.1) : A lexical token that appears in an INTERFACE statement and is  
26 associated with all the procedures in the interface block or that appears in a GENERIC statement and  
27 is associated with the specific type-bound procedures.
- 28 **generic interface** (4.5.2, 12.4.2.1) : An interface specified by a generic procedure binding or a generic  
29 interface block.
- 30 **generic interface block** (12.4.2.1) : An interface block with a generic specification.
- 31 **global entity** (16.2) : An entity with an identifier whose scope is a program.
- 32 **host** (2.2) : Host scoping unit.
- 33 **host association** (16.5.1.4) : The process by which a contained scoping unit accesses entities of its  
34 host.
- 35 **host scoping unit** (2.2) : A scoping unit that immediately surrounds another scoping unit.
- 36 **image** (2.3.5) : A Fortran program executes as if it were replicated a number of times, the number of  
37 replications remaining fixed during execution of the program. Each copy is called an **image** and each

- 1 image executes asynchronously.
- 2 **image index** (2.3.5) : An integer value that identifies an image.

### J3 internal note

Unresolved Technical Issue 78

Clause 16 is missing edits about images. (And so is Annex A.) In particular:

- images are entities – add them to the entity gloss;
- they are identified by integer image indexes – add that fact to the first numbered list in clause 16;
- they are global entities – this should be noted at the end of “Scope of global identifiers” (was 16.1).

While I’m here, I can well imagine an obvious implementation technique for co-arrays/images on shared-memory machines which would result in pending data transfer operations being per-program not per-image. I’ve not been back to check to see whether this is relevant to the current set of things one is allowed to do to a connect-team file, though I can well imagine that it might be (once the rules are sorted out).

Finally, there are very real questions (hinted at if not raised elsewhere) about the adequacy of our simple local+global scoping model w.r.t. co-arrays. Maybe this needs to have three levels rather than two?

- 3 **implicit interface** (12.4.1) : For a procedure referenced in a scoping unit, the property of not having
- 4 an explicit interface. A statement function always has an implicit interface
- 5 **inherit** (4.5.7) : To acquire from a parent. Type parameters, components, or procedure bindings of an
- 6 extended type that are automatically acquired from its parent type without explicit declaration in the
- 7 extended type are said to be inherited.
- 8 **inheritance association** (4.5.7.2, 16.5.4) : The relationship between the inherited components and the
- 9 parent component in an extended type.
- 10 **inquiry function** (13.1) : A function that is either intrinsic or is defined in an intrinsic module and
- 11 whose result depends on properties of one or more of its arguments instead of their values.
- 12 **instance of a subprogram** (12.6.2.3) : The copy of a subprogram that is created when a procedure
- 13 defined by the subprogram is invoked.
- 14 **intent** (5.3.9) : An attribute of a dummy data object that indicates whether it is used to transfer data
- 15 into the procedure, out of the procedure, or both.
- 16 **interface block** (12.4.2.1) : A sequence of statements from an INTERFACE statement to the corre-
- 17 sponding END INTERFACE statement.
- 18 **interface body** (12.4.2.1) : A sequence of statements in an interface block from a FUNCTION or
- 19 SUBROUTINE statement to the corresponding END statement.
- 20 **interface of a procedure** (12.4) : See procedure interface.
- 21 **internal file** (9.3) : A character variable that is used to transfer and convert data from internal storage
- 22 to internal storage.
- 23 **internal procedure** (2.2.3.3) : A procedure that is defined by an internal subprogram.
- 24 **internal subprogram** (2.2) : A subprogram in a main program or another subprogram.

- 1 **interoperable** (15.3) : The property of a Fortran entity that ensures that an equivalent entity may be  
2 defined by means of C.
- 3 **intrinsic** (2.5.7) : An adjective that may be applied to types, operators, assignment statements, proce-  
4 dures, and modules. Intrinsic types, operators, and assignment statements are defined in this standard  
5 and may be used in any scoping unit without further definition or specification. Intrinsic procedures are  
6 defined in this standard or provided by a processor, and may be used in a scoping unit without further  
7 definition or specification. Intrinsic modules are defined in this standard or provided by a processor,  
8 and may be accessed by use association; procedures and types defined in an intrinsic module are not  
9 themselves intrinsic.
- 10 Intrinsic procedures and modules that are not defined in this standard are called nonstandard intrinsic  
11 procedures and modules.
- 12 **invoke** (2.2.3) :
- 13 To call a subroutine by a CALL statement or by a defined assignment statement.
- 14 To call a function by a reference to it by name or operator during the evaluation of an expression.
- 15 To call a final subroutine by finalization.
- 16 **keyword** (2.5.2) : A word that is part of the syntax of a statement or a name that is used to identify  
17 an item in a list.
- 18 **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  
19 the available kinds of an intrinsic type, or a derived-type parameter that is declared to have the KIND  
20 attribute.
- 21 **label** : See binding label or statement label.
- 22 **length of a character string** (4.4.5) : The number of characters in the character string.
- 23 **lexical token** (3.2) : A sequence of one or more characters with a specified interpretation.
- 24 **line** (3.3) : A sequence of 0 to 132 characters, which may contain Fortran statements, a comment, or  
25 an INCLUDE line.
- 26 **linkage association** (16.5.1.5) : The association between interoperable Fortran entities and their C  
27 counterparts.
- 28 **literal constant** (2.4.3.1.2, 4.4) : A constant without a name. Note that in FORTRAN 77, this was  
29 called simply a constant.
- 30 **local entity** (16.3) : An entity identified by a lexical token whose scope is a scoping unit.
- 31 **local variable** (2.4.3.1.1) : A variable local to a particular scoping unit; not imported through use or  
32 host association, not a dummy argument, and not a variable in common.
- 33 **main program** (2.3.4, 11.1) : A Fortran main program or a replacement defined by means other than  
34 Fortran.
- 35 **many-one array section** (6.2.2.3.2) : An array section with a vector subscript having two or more  
36 elements with the same value.
- 37 **module** (2.2.4, 11.2) : A program unit that contains or accesses definitions to be accessed by other  
38 program units.
- 39 **module procedure** (2.2.3.2) : 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.

- 1 **module procedure interface** (12.4.2.1) : The interface for a separate module procedure.
- 2 **module subprogram** (2.2) : A subprogram that is in a module but is not an internal subprogram.
- 3 **name** (3.2.1) : A lexical token consisting of a letter followed by up to 62 alphanumeric characters
- 4 (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.
- 5 **name association** (16.5.1) : Argument association, use association, host association, linkage associa-
- 6 tion, or construct association.
- 7 **named** : Having a name. That is, in a phrase such as “named variable,” the word “named” signifies that
- 8 the variable name is not qualified by a subscript list, substring specification, and so on. For example,
- 9 if X is an array variable, the reference “X” is a named variable while the reference “X(1)” is an object
- 10 designator.
- 11 **named constant** (2.4.3.1.2) : A constant that has a name. Note that in FORTRAN 77, this was called
- 12 a symbolic constant.
- 13 **NaN** (14.7) : A Not-a-Number value of IEEE arithmetic. It represents an undefined value or a value
- 14 created by an invalid operation.
- 15 **nonexecutable statement** (2.3.1) : A statement used to configure the program environment in which
- 16 computational actions take place.
- 17 **numeric storage unit** (16.5.3.2) : The unit of storage for holding a scalar that is not a pointer and is
- 18 of type default real, default integer, or default logical.
- 19 **numeric type** (4.4) : Integer, real or complex type.
- 20 **object** (2.4.3.1) : Data object.
- 21 **object designator** (2.5.1) : A designator for a data object.
- 22 **obsolescent feature** (1.8) : A feature that is considered to have been redundant but that is still in
- 23 frequent use.
- 24 **operand** (2.5.8) : An expression that precedes or succeeds an operator.
- 25 **operation** (7.1.2) : A computation involving one or two operands.
- 26 **operator** (2.5.8) : A lexical token that specifies an operation.
- 27 **override** (4.5.2, 4.5.7) : When explicit initialization or default initialization overrides default initializa-
- 28 tion, it is as if only the overriding initialization were specified. If a procedure is bound to an extensible
- 29 type, it overrides the one that would have been inherited from the parent type.
- 30 **parent component** (4.5.7.2) : The component of an entity of extended type that corresponds to its
- 31 inherited portion.

- 1 **parent type** (4.5.7) : The extensible type from which an extended type is derived.
- 2 **passed-object dummy argument** (4.5.4.4) : The dummy argument of a type-bound procedure or  
3 procedure pointer component that becomes associated with the object through which the procedure was  
4 invoked.
- 5 **pointer** (2.4.7) : An entity that has the POINTER attribute.
- 6 **pointer assignment** (7.4.2) : The pointer association of a pointer with a target by the execution of a  
7 pointer assignment statement or the execution of an assignment statement for a data object of derived  
8 type having the pointer as a subobject.
- 9 **pointer assignment statement** (7.4.2) : A statement of the form “pointer-object => target”.
- 10 **pointer associated** (6.3, 7.4.2) : The relationship between a pointer and a target following a pointer  
11 assignment or a valid execution of an ALLOCATE statement.
- 12 **pointer association** (16.5.2) : The process by which a pointer becomes pointer associated with a target.
- 13 **polymorphic** (4.3.1.3) : Able to be of differing types during program execution. An object declared  
14 with the CLASS keyword is polymorphic.
- 15 **preconnected** (9.4.4) : A property describing a unit that is connected to an external file at the beginning  
16 of execution of a program. Such a unit may be specified in input/output statements without an OPEN  
17 statement being executed for that unit.
- 18 **procedure** (2.2.3, 12.2) : A computation that may be invoked during program execution. It may be a  
19 function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure,  
20 an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than  
21 one procedure if it contains ENTRY statements.
- 22 **procedure designator** (2.5.1) : A designator for a procedure.
- 23 **procedure interface** (12.4) : The characteristics of a procedure, the name of the procedure, the name  
24 of each dummy argument, and the generic identifiers (if any) by which it may be referenced.
- 25 **processor** (1.2) : The combination of a computing system and the mechanism by which programs are  
26 transformed for use on that computing system.
- 27 **processor dependent** (1.5) : The designation given to a facility that is not completely specified by  
28 this standard. Such a facility shall be provided by a processor, with methods or semantics determined  
29 by the processor.
- 30 **program** (2.2.1) : A set of program units that includes exactly one main program.
- 31 **program unit** (2.2) : The fundamental component of a program. A sequence of statements, comments,  
32 and INCLUDE lines. It may be a main program, a module, an external subprogram, or a block data  
33 program unit.
- 34 **prototype** : The C analog of a function interface body; defined in 6.7.5.3 of the C International  
35 Standard.
- 36 **pure procedure** (12.7) : A procedure that is a pure intrinsic procedure (13.1), is defined by a pure  
37 subprogram, or is a statement function that references only pure functions.
- 38 **rank** (2.4.4, 2.4.5) : The number of dimensions of an array. Zero for a scalar.
- 39 **record** (9.1) : A sequence of values or characters that is treated as a whole within a file.



- 1 **reference** (2.5.6) : The appearance of an object designator in a context requiring the value at that  
2 point during execution, the appearance of a procedure designator, its operator symbol, or a defined  
3 assignment statement in a context requiring execution of the procedure at that point, or the appearance  
4 of a module name in a USE statement. Neither the act of defining a variable nor the appearance of the  
5 name of a procedure as an actual argument is regarded as a reference.
- 6 **result variable** (2.2.3, 12.6.2.1) : The variable that returns the value of a function.
- 7 **rounding mode** (14.3, 10.7.2.3.7) : The method used to choose the result of an operation that cannot  
8 be represented exactly. In IEEE arithmetic, there are four modes; nearest, towards zero, up (towards  
9  $\infty$ ), and down (towards  $-\infty$ ). In addition, for input/output the two additional modes COMPATIBLE  
10 and PROCESSOR\_DEFINED are provided.
- 11 **scalar** (2.4.4) :  
12 Not being an array.
- 13 **scope** (16) : That part of a program within which a lexical token has a single interpretation. It may be  
14 a program, a scoping unit, a construct, a single statement, or a part of a statement.
- 15 **scoping unit** (2.2) : One of the following:  
16 (1) A program unit or subprogram, excluding any scoping units in it,  
17 (2) A derived-type definition, or  
18 (3) An interface body, excluding any scoping units in it.
- 19 **section subscript** (6.2.2) : A subscript, vector subscript, or subscript triplet in an array section selector.
- 20 **selector** (6.1.1, 6.1.2, 6.1.4, 8.1.4, 8.1.5) : A syntactic mechanism for designating  
21 (1) a subobject,  
22 (2) the set of values for which a CASE block is executed,  
23 (3) the object whose type determines which branch of a SELECT TYPE construct is executed,  
24 or  
25 (4) the object that is associated with the *associate-name* in an ASSOCIATE construct.
- 26 **shape** (2.4.5) : The rank and extents of an array. The shape may be represented by the rank-one array  
27 whose elements are the extents in each dimension.
- 28 **size** (2.4.5) : The total number of elements of an array.
- 29 **specification expression** (7.1.6) : An expression with limitations that make it suitable for use in  
30 specifications such as length type parameters or array bounds.
- 31 **specification function** (7.1.6) : A nonintrinsic function that may be used in a specification expression.
- 32 **standard-conforming program** (1.5) : A program that uses only those forms and relationships de-  
33 scribed in this standard, and that has an interpretation according to this standard.
- 34 **statement** (3.3) : A sequence of lexical tokens. It usually consists of a single line, but a statement may  
35 be continued from one line to another and the semicolon symbol may be used to separate statements  
36 within a line.
- 37 **statement entity** (16) : An entity identified by a lexical token whose scope is a single statement or  
38 part of a statement.
- 39 **statement function** (12.6.4) : A procedure specified by a single statement that is similar in form to an assignment  
40 statement.

- 1 **statement label** (3.2.4) : A lexical token consisting of up to five digits that precedes a statement and  
2 may be used to refer to the statement.
- 3 **storage association** (16.5.3) : The relationship between two storage sequences if a storage unit of one  
4 is the same as a storage unit of the other.
- 5 **storage sequence** (16.5.3.2) : A sequence of contiguous storage units.
- 6 **storage unit** (16.5.3.2) : A character storage unit, a numeric storage unit, a file storage unit, or an  
7 unspecified storage unit.
- 8 **stride** (6.2.2.3.1) : The increment specified in a subscript triplet.
- 9 **struct** : The C analog of a sequence derived type; defined in 6.2.5 of the C International Standard.
- 10 **structure** (2.4.1.2) : A scalar data object of derived type.
- 11 **structure component** (6.1.2) : A part of an object of derived type. It may be referenced by an object  
12 designator.
- 13 **structure constructor** (4.5.10) : A syntactic mechanism for constructing a value of derived type.
- 14 **subcomponent** (6.1.2) : A subcomponent of an object of derived type is a component of that object  
15 or of a subobject of that object.
- 16 **submodule** (2.2.5, 11.2.2) : A program unit that extends a module or another submodule.
- 17 **subobject** (2.4.3.1) : A portion of a data object that may be referenced or defined independently of  
18 other portions. It may be an array element, an array section, a structure component, a substring, or the  
19 real or imaginary part of a complex object.
- 20 **subprogram** (2.2) : A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a  
21 block data program unit was called a subprogram.
- 22 **subroutine** (2.2.3) : A procedure that is invoked by a CALL statement or by a defined assignment  
23 statement.
- 24 **subroutine subprogram** (12.6.2.2) : A sequence of statements beginning with a SUBROUTINE state-  
25 ment that is not in an interface block and ending with the corresponding END statement.
- 26 **subscript** (6.2.2) : One of the list of scalar integer expressions in an array element selector. Note that  
27 in FORTRAN 77, the whole list was called the subscript.
- 28 **subscript triplet** (6.2.2) : An item in the list of an array section selector that contains a colon and  
29 specifies a regular sequence of integer values.
- 30 **substring** (6.1.1) : A contiguous portion of a scalar character string. Note that an array section can  
31 include a substring selector; the result is called an array section and not a substring.
- 32 **target** (2.4.7, 5.3.16, 6.3.1.2) : A data entity that has the TARGET attribute, or an entity that is  
33 associated with a pointer.
- 34 **team** (2.3.5) : A set of images formed by invoking the intrinsic collective subroutine FORM\_TEAM  
35 (13.7.69).
- 36 **team synchronization** (8.5.3) : Synchronization of the images in a team.
- 37 **transformational function** (13.1) : A function that is either intrinsic or is defined in an intrinsic  
38 module and that is neither an elemental function nor an inquiry function.

- 1 **type** (2.4.1) : A named category of data that is characterized by a set of values, together with a way to  
2 denote these values and a collection of operations that interpret and manipulate the values. The set of  
3 data values depends on the values of the type parameters.
- 4 **type-bound procedure** (4.5.5) : A procedure binding in a type definition. The procedure may be  
5 referenced by the *binding-name* via any object of that dynamic type, as a defined operator, by defined  
6 assignment, or as part of the finalization process.
- 7 **type compatible** (4.3.1.3) : All entities are type compatible with other entities of the same type.  
8 Unlimited polymorphic entities are type compatible with all entities; other polymorphic entities are type  
9 compatible with entities whose dynamic type is an extension type of the polymorphic entity's declared  
10 type.
- 11 **type declaration statement** (5.2) : An INTEGER, REAL, DOUBLE PRECISION, COMPLEX,  
12 CHARACTER, LOGICAL, TYPE (*type-name*), or CLASS (*type-name*) statement.
- 13 **type parameter** (2.4.1.1) : A parameter of a data type. KIND and LEN are the type parameters of  
14 intrinsic types. The type parameters of a derived type are defined in the derived-type definition.
- 15 **type parameter order** (4.5.3.2) : The ordering of the type parameters of a derived type that is used  
16 for derived-type specifiers.
- 17 **ultimate component** (4.5) : For a structure, a component that is of intrinsic type, has the ALLOCAT-  
18 ABLE attribute, or has the POINTER attribute, or an ultimate component of a derived-type component  
19 that does not have the POINTER attribute or the ALLOCATABLE attribute.
- 20 **undefined** (2.5.5) : For a data object, the property of not having a determinate value.
- 21 **unsigned** : A qualifier of a C numeric type indicating that it is comprised only of nonnegative values;  
22 defined in 6.2.5 of the C International Standard. There is nothing analogous in Fortran.
- 23 **unspecified storage unit** (16.5.3.2) : A unit of storage for holding a pointer or a scalar that is not a  
24 pointer and is of type other than default integer, default character, default real, double precision real,  
25 default logical, or default complex.
- 26 **use association** (16.5.1.3) : The association of names in different scoping units specified by a USE  
27 statement.
- 28 **variable** (2.4.3.1.1) : A data object whose value can be defined and redefined during the execution of  
29 a program. It may be a named data object, an array element, an array section, a structure component,  
30 or a substring. Note that in FORTRAN 77, a variable was always scalar and named.
- 31 **vector subscript** (6.2.2.3.2) : A section subscript that is an integer expression of rank one.
- 32 **void** : A C type comprising an empty set of values; defined in 6.2.5 of the C International Standard.  
33 There is nothing analogous in Fortran.
- 34 **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 standard are the following.

- 5 (1) Arithmetic IF — use the IF statement (8.1.3.4) or IF construct (8.1.3).
- 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  y = t(42,mold=y)           ! References int_to_t2
2  y = t(kind(0d0)) ((0,1))  ! Uses the structure constructor for type t
3  END

```

#### 4 C.1.6 Generic type-bound procedures

5 Example of a derived type with generic type-bound procedures:

6 The only difference between this example and the same thing rewritten to use generic interface blocks  
7 is that with type-bound procedures,

```

8      USE(rational_numbers),ONLY :: rational

```

9 does not block the type-bound procedures; the user still gets access to the defined assignment and  
10 extended operations.

```

11 MODULE rational_numbers
12   IMPLICIT NONE
13   PRIVATE
14   TYPE,PUBLIC :: rational
15     PRIVATE
16     INTEGER n,d
17   CONTAINS
18     ! ordinary type-bound procedure
19     PROCEDURE :: real => rat_to_real
20     ! specific type-bound procedures for generic support
21     PROCEDURE,PRIVATE :: rat_asgn_i
22     PROCEDURE,PRIVATE :: rat_plus_rat
23     PROCEDURE,PRIVATE :: rat_plus_i
24     PROCEDURE,PRIVATE,PASS(b) :: i_plus_rat
25     ! generic type-bound procedures
26     GENERIC :: ASSIGNMENT(=) => rat_asgn_i
27     GENERIC :: OPERATOR(+) => rat_plus_rat, rat_plus_i, i_plus_rat
28   END TYPE
29 CONTAINS
30   ELEMENTAL REAL FUNCTION rat_to_real(this) RESULT(r)
31     CLASS(rational),INTENT(IN) :: this
32     r = REAL(this%n)/this%d
33   END FUNCTION
34   ELEMENTAL SUBROUTINE rat_asgn_i(a,b)
35     CLASS(rational),INTENT(OUT) :: a
36     INTEGER,INTENT(IN) :: b
37     a%n = b
38     a%d = 1
39   END SUBROUTINE
40   ELEMENTAL TYPE(rational) FUNCTION rat_plus_i(a,b) RESULT(r)

```

```

1      CLASS(rational),INTENT(IN) :: a
2      INTEGER,INTENT(IN) :: b
3      r%n = a%n + b*a%d
4      r%d = a%d
5  END FUNCTION
6  ELEMENTAL TYPE(rational) FUNCTION i_plus_rat(a,b) RESULT(r)
7      INTEGER,INTENT(IN) :: a
8      CLASS(rational),INTENT(IN) :: b
9      r%n = b%n + a*b%d
10     r%d = b%d
11 END FUNCTION
12 ELEMENTAL TYPE(rational) FUNCTION rat_plus_rat(a,b) RESULT(r)
13     CLASS(rational),INTENT(IN) :: a,b
14     r%n = a%n*b%d + b%n*a%d
15     r%d = a%d*b%d
16 END FUNCTION
17 END

```

### 18 C.1.7 Final subroutines (4.5.6, 4.5.6.2, 4.5.6.3, 4.5.6.4)

19 Example of a parameterized derived type with final subroutines:

```

20 MODULE m
21     TYPE t(k)
22         INTEGER, KIND :: k
23         REAL(k),POINTER :: vector(:) => NULL()
24     CONTAINS
25         FINAL :: finalize_t1s, finalize_t1v, finalize_t2e
26     END TYPE
27 CONTAINS
28     SUBROUTINE finalize_t1s(x)
29         TYPE(t(KIND(0.0))) x
30         IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
31     END SUBROUTINE
32     SUBROUTINE finalize_t1v(x)
33         TYPE(t(KIND(0.0))) x(:)
34         DO i=LBOUND(x,1),UBOUND(x,1)
35             IF (ASSOCIATED(x(i)%vector)) DEALLOCATE(x(i)%vector)
36         END DO
37     END SUBROUTINE
38     ELEMENTAL SUBROUTINE finalize_t2e(x)
39         TYPE(t(KIND(0.0d0))),INTENT(INOUT) :: x
40         IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
41     END SUBROUTINE
42 END MODULE

```

```
1
2 SUBROUTINE example(n)
3   USE m
4   TYPE(t(KIND(0.0))) a,b(10),c(n,2)
5   TYPE(t(KIND(0.0d0))) d(n,n)
6   ...
7   ! Returning from this subroutine will effectively do
8   !   CALL finalize_t1s(a)
9   !   CALL finalize_t1v(b)
10  !   CALL finalize_t2e(d)
11  ! No final subroutine will be called for variable C because the user
12  ! omitted to define a suitable specific procedure for it.
13 END SUBROUTINE
```

14 Example of extended types with final subroutines:

```
15 MODULE m
16   TYPE t1
17     REAL a,b
18   END TYPE
19   TYPE,EXTENDS(t1) :: t2
20     REAL,POINTER :: c(:),d(:)
21   CONTAINS
22     FINAL :: t2f
23   END TYPE
24   TYPE,EXTENDS(t2) :: t3
25     REAL,POINTER :: e
26   CONTAINS
27     FINAL :: t3f
28   END TYPE
29   ...
30 CONTAINS
31   SUBROUTINE t2f(x) ! Finalizer for TYPE(t2)'s extra components
32     TYPE(t2) :: x
33     IF (ASSOCIATED(x%c)) DEALLOCATE(x%c)
34     IF (ASSOCIATED(x%d)) DEALLOCATE(x%d)
35   END SUBROUTINE
36   SUBROUTINE t3f(y) ! Finalizer for TYPE(t3)'s extra components
37     TYPE(t3) :: y
38     IF (ASSOCIATED(y%e)) DEALLOCATE(y%e)
39   END SUBROUTINE
40 END MODULE
41
42 SUBROUTINE example
43   USE m
```



```

1  TYPE(t1) x1
2  TYPE(t2) x2
3  TYPE(t3) x3
4  ...
5  ! Returning from this subroutine will effectively do
6  !      ! Nothing to x1; it is not finalizable
7  !      CALL t2f(x2)
8  !      CALL t3f(x3)
9  !      CALL t2f(x3%t2)
10 END SUBROUTINE

```

## 11 C.2 Clause 5 notes

### 12 C.2.1 The POINTER attribute (5.3.13)

13 The POINTER attribute shall be specified to declare a pointer. The type, type parameters, and rank,  
 14 which may be specified in the same statement or with one or more attribute specification statements,  
 15 determine the characteristics of the target objects that may be associated with the pointers declared  
 16 in the statement. An obvious model for interpreting declarations of pointers is that such declarations  
 17 create for each name a descriptor. Such a descriptor includes all the data necessary to describe fully  
 18 and locate in memory an object and all subobjects of the type, type parameters, and rank specified.  
 19 The descriptor is created empty; it does not contain values describing how to access an actual memory  
 20 space. These descriptor values will be filled in when the pointer is associated with actual target space.

21 The following example illustrates the use of pointers in an iterative algorithm:

```

22 PROGRAM DYNAM_ITER
23   REAL, DIMENSION (:, :), POINTER :: A, B, SWAP ! Declare pointers
24   ...
25   READ (*, *) N, M
26   ALLOCATE (A (N, M), B (N, M)) ! Allocate target arrays
27   ! Read values into A
28   ...
29   ITER: DO
30     ...
31     ! Apply transformation of values in A to produce values in B
32     ...
33     IF (CONVERGED) EXIT ITER
34     ! Swap A and B
35     SWAP => A; A => B; B => SWAP
36   END DO ITER
37   ...
38 END PROGRAM DYNAM_ITER

```

## 1 C.2.2 The TARGET attribute (5.3.16)

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  PRINT *, MARY % ID % LAST_NAME      ! Print LAST_NAME of MARY
2  PRINT *, MARY % ID % NUMBER % SSN ! Print SSN of MARY
3  PRINT *, GEORGE % ID % NUMBER      ! Print SSN and EMPLOYEE_NUMBER of GEORGE

```

4 A structure component may be a data object of intrinsic type as in the case of GEORGE % AGE or it  
 5 may be of derived type as in the case of GEORGE % ID % NUMBER. The resultant component may  
 6 be a scalar or an array of intrinsic or derived type.

```

7  TYPE LARGE
8      INTEGER ELT (10)
9      INTEGER VAL
10 END TYPE LARGE
11
12 TYPE (LARGE) A (5)          ! 5 element array, each of whose elements
13                             ! includes a 10 element array ELT and
14                             ! a scalar VAL.
15 PRINT *, A (1)              ! Prints 10 element array ELT and scalar VAL.
16 PRINT *, A (1) % ELT (3)    ! Prints scalar element 3
17                             ! of array element 1 of A.
18 PRINT *, A (2:4) % VAL      ! Prints scalar VAL for array elements
19                             ! 2 to 4 of A.

```

20 Components of an object of extensible type that are inherited from the parent type may be accessed as  
 21 a whole by using the parent component name, or individually, either with or without qualifying them  
 22 by the parent component name.

23 For example:

```

24  TYPE POINT                ! A base type
25      REAL :: X, Y
26  END TYPE POINT
27  TYPE, EXTENDS(POINT) :: COLOR_POINT ! An extension of TYPE(POINT)
28      ! Components X and Y, and component name POINT, inherited from parent
29      INTEGER :: COLOR
30  END TYPE COLOR_POINT
31
32  TYPE(POINT) :: PV = POINT(1.0, 2.0)
33  TYPE(COLOR_POINT) :: CPV = COLOR_POINT(PV, 3) ! Nested form constructor
34
35  PRINT *, CPV%POINT          ! Prints 1.0 and 2.0
36  PRINT *, CPV%POINT%X, CPV%POINT%Y ! And this does, too
37  PRINT *, CPV%X, CPV%Y      ! And this does, too

```

### 38 C.3.2 Allocation with dynamic type (6.3.1)

39 The following example illustrates the use of allocation with the value and dynamic type of the allocated  
 40 object given by another object. The example copies a list of objects of any type. It copies the list

1 starting at IN\_LIST. After copying, each element of the list starting at LIST\_COPY has a polymorphic  
 2 component, ITEM, for which both the value and type are taken from the ITEM component of the  
 3 corresponding element of the list starting at IN\_LIST.

```

4 TYPE :: LIST ! A list of anything
5   TYPE(LIST), POINTER :: NEXT => NULL()
6   CLASS(*), ALLOCATABLE :: ITEM
7 END TYPE LIST
8 ...
9 TYPE(LIST), POINTER :: IN_LIST, LIST_COPY => NULL()
10 TYPE(LIST), POINTER :: IN_WALK, NEW_TAIL
11 ! Copy IN_LIST to LIST_COPY
12 IF (ASSOCIATED(IN_LIST)) THEN
13   IN_WALK => IN_LIST
14   ALLOCATE(LIST_COPY)
15   NEW_TAIL => LIST_COPY
16   DO
17     ALLOCATE(NEW_TAIL%ITEM, SOURCE=IN_WALK%ITEM)
18     IN_WALK => IN_WALK%NEXT
19     IF (.NOT. ASSOCIATED(IN_WALK)) EXIT
20     ALLOCATE(NEW_TAIL%NEXT)
21     NEW_TAIL => NEW_TAIL%NEXT
22   END DO
23 END IF

```

### 24 C.3.3 Pointer allocation and association

25 The effect of ALLOCATE, DEALLOCATE, NULLIFY, and pointer assignment is that they are inter-  
 26 preted as changing the values in the descriptor that is the pointer. An ALLOCATE is assumed to create  
 27 space for a suitable object and to “assign” to the pointer the values necessary to describe that space.  
 28 A NULLIFY breaks the association of the pointer with the space. A DEALLOCATE breaks the asso-  
 29 ciation and releases the space. Depending on the implementation, it could be seen as setting a flag in  
 30 the pointer that indicates whether the values in the descriptor are valid, or it could clear the descriptor  
 31 values to some (say zero) value indicative of the pointer not pointing to anything. A pointer assignment  
 32 copies the values necessary to describe the space occupied by the target into the descriptor that is the  
 33 pointer. Descriptors are copied, values of objects are not.

34 If PA and PB are both pointers and PB is associated with a target, then

35 PA => PB

36 results in PA being associated with the same target as PB. If PB was disassociated, then PA becomes  
 37 disassociated.

38 The standard is specified so that such associations are direct and independent. A subsequent statement

39 PB => D

40 or

41 ALLOCATE (PB)

1 has no effect on the association of PA with its target. A statement

2 DEALLOCATE (PB)

3 deallocates the space that is associated with both PA and PB. PB becomes disassociated, but there is  
 4 no requirement that the processor make it explicitly recognizable that PA no longer has a target. This  
 5 leaves PA as a “dangling pointer” to space that has been released. The program shall not use PA again  
 6 until it becomes associated via pointer assignment or an ALLOCATE statement.

7 DEALLOCATE may only be used to release space that was created by a previous ALLOCATE. Thus  
 8 the following is invalid:

9 REAL, TARGET :: T

10 REAL, POINTER :: P

11 ...

12 P = > T

13 DEALLOCATE (P) ! Not allowed: P's target was not allocated

14 The basic principle is that ALLOCATE, NULLIFY, and pointer assignment primarily affect the pointer  
 15 rather than the target. ALLOCATE creates a new target but, other than breaking its connection with  
 16 the specified pointer, it has no effect on the old target. Neither NULLIFY nor pointer assignment has  
 17 any effect on targets. A piece of memory that was allocated and associated with a pointer will become  
 18 inaccessible to a program if the pointer is nullified or associated with a different target and no other  
 19 pointer was associated with this piece of memory. Such pieces of memory may be reused by the processor  
 20 if this is expedient. However, whether such inaccessible memory is in fact reused is entirely processor  
 21 dependent.

## 22 C.4 Clause 7 notes

### 23 C.4.1 Character assignment

24 The FORTRAN 77 restriction that none of the character positions being defined in the character assign-  
 25 ment statement may be referenced in the expression has been removed (7.4.1.3).

### 26 C.4.2 Evaluation of function references

27 If more than one function reference appears in a statement, they may be executed in any order (subject to  
 28 a function result being evaluated after the evaluation of its arguments) and their values shall not depend  
 29 on the order of execution. This lack of dependence on order of evaluation permits parallel execution of  
 30 the function references (7.1.8.1).

### 31 C.4.3 Pointers in expressions

32 A pointer is basically considered to be like any other variable when it is used as a primary in an expression.  
 33 If a pointer is used as an operand to an operator that expects a value, the pointer will automatically  
 34 deliver the value stored in the space described by the pointer, that is, the value of the target object  
 35 associated with the pointer.

### 36 C.4.4 Pointers on the left side of an assignment

37 A pointer that appears on the left of an intrinsic assignment statement also is dereferenced and is taken  
 38 to be referring to the space that is its current target. Therefore, the assignment statement specifies the

1 normal copying of the value of the right-hand expression into this target space. All the normal rules of  
 2 intrinsic assignment hold; the type and type parameters of the expression and the pointer target shall  
 3 agree and the shapes shall be conformable.

4 For intrinsic assignment of derived types, nonpointer components are assigned and pointer components  
 5 are pointer assigned. Dereferencing is applied only to entire scalar objects, not selectively to pointer  
 6 subobjects.

7 For example, suppose a type such as

```
8 TYPE CELL
9     INTEGER :: VAL
10     TYPE (CELL), POINTER :: NEXT_CELL
11 END TYPE CELL
```

12 is defined and objects such as HEAD and CURRENT are declared using

```
13 TYPE (CELL), TARGET :: HEAD
14 TYPE (CELL), POINTER :: CURRENT
```

15 If a linked list has been created and attached to HEAD and the pointer CURRENT has been allocated  
 16 space, statements such as

```
17 CURRENT = HEAD
18 CURRENT = CURRENT % NEXT_CELL
```

19 cause the contents of the cells referenced on the right to be copied to the cell referred to by CURRENT.  
 20 In particular, the right-hand side of the second statement causes the pointer component in the cell,  
 21 CURRENT, to be selected. This pointer is dereferenced because it is in an expression context to produce  
 22 the target's integer value and a pointer to a cell that is in the target's NEXT\_CELL component. The  
 23 left-hand side causes the pointer CURRENT to be dereferenced to produce its present target, namely  
 24 space to hold a cell (an integer and a cell pointer). The integer value on the right is copied to the integer  
 25 space on the left and the pointer component is pointer assigned (the descriptor on the right is copied  
 26 into the space for a descriptor on the left). When a statement such as

```
27 CURRENT => CURRENT % NEXT_CELL
```

28 is executed, the descriptor value in CURRENT % NEXT\_CELL is copied to the descriptor named  
 29 CURRENT. In this case, CURRENT is made to point at a different target.

30 In the intrinsic assignment statement, the space associated with the current pointer does not change but  
 31 the values stored in that space do. In the pointer assignment, the current pointer is made to associate  
 32 with different space. Using the intrinsic assignment causes a linked list of cells to be moved up through  
 33 the current "window"; the pointer assignment causes the current pointer to be moved down through the  
 34 list of cells.

### 35 **C.4.5 An example of a FORALL construct containing a WHERE construct**

```
36 INTEGER :: A(5,5)
37 ...
38 FORALL (I = 1:5)
39     WHERE (A(I,:) == 0)
```

```

1      A(:,I) = I
2      ELSEWHERE (A(I,:) > 2)
3      A(I,:) = 6
4      END WHERE
5  END FORALL

```

6 If prior to execution of the FORALL, A has the value

```

7  A =      1  0  0  0  0
8           2  1  1  1  0
9           1  2  2  0  2
10          2  1  0  2  3
11          1  0  0  0  0

```

12 After execution of the assignment statements following the WHERE statement A has the value A'. The  
 13 mask created from row one is used to mask the assignments to column one; the mask from row two is  
 14 used to mask assignments to column two; etc.

```

15 A' =      1  0  0  0  0
16           1  1  1  1  5
17           1  2  2  4  5
18           1  1  3  2  5
19           1  2  0  0  5

```

20 The masks created for assignments following the ELSEWHERE statement are

```

21 .NOT. (A(I,:) == 0) .AND. (A'(I,:) > 2)

```

22 Thus the only elements affected by the assignments following the ELSEWHERE statement are A(3, 5)  
 23 and A(4, 5). After execution of the FORALL construct, A has the value

```

24 A =      1  0  0  0  0
25           1  1  1  1  5
26           1  2  2  4  6
27           1  1  3  2  6
28           1  2  0  0  5

```

## 29 C.4.6 Examples of FORALL statements

30 Example 1:

```

31 FORALL (J=1:M, K=1:N) X(K, J) = Y(J, K)
32 FORALL (K=1:N) X(K, 1:M) = Y(1:M, K)

```

33 These statements both copy columns 1 through N of array Y into rows 1 through N of array X. They  
 34 are equivalent to

```

35 X(1:N, 1:M) = TRANSPOSE (Y(1:M, 1:N) )

```



1 Example 2:

2 The following FORALL statement computes five partial sums of subarrays of J.

3 J = (/ 1, 2, 3, 4, 5 /)

4 FORALL (K = 1:5) J(K) = SUM (J(1:K) )

5 SUM is allowed in a FORALL because intrinsic functions are pure (12.7). After execution of the FORALL  
6 statement, J = (/ 1, 3, 6, 10, 15 /).

7 Example 3:

8 FORALL (I = 2:N-1) X(I) = (X(I-1) + 2\*X(I) + X(I+1) ) / 4

9 has the same effect as

10 X(2:N-1) = (X(1:N-2) + 2\*X(2:N-1) + X(3:N) ) / 4

## 11 C.5 Clause 8 notes

### 12 C.5.1 Loop control

13 Fortran provides several forms of loop control:

14 (1) With an iteration count and a DO variable. This is the classic Fortran DO loop.

15 (2) Test a logical condition before each execution of the loop (DO WHILE).

16 (3) DO “forever”.

### 17 C.5.2 The CASE construct

18 At most one case block is selected for execution within a CASE construct, and there is no fall-through  
19 from one block into another block within a CASE construct. Thus there is no requirement for the user  
20 to exit explicitly from a block.

### 21 C.5.3 Examples of DO constructs

22 The following are all valid examples of block DO constructs.

23 Example 1:

```

24     SUM = 0.0
25     READ (IUN) N
26     OUTER: DO L = 1, N                ! A DO with a construct name
27         READ (IUN) IQUAL, M, ARRAY (1:M)
28         IF (IQUAL < IQUAL_MIN) CYCLE OUTER    ! Skip inner loop
29         INNER: DO 40 I = 1, M            ! A DO with a label and a name
30             CALL CALCULATE (ARRAY (I), RESULT)
31             IF (RESULT < 0.0) CYCLE
32             SUM = SUM + RESULT
33             IF (SUM > SUM_MAX) EXIT OUTER
34 40     END DO INNER
35     END DO OUTER

```

1 The outer loop has an iteration count of MAX (N, 0), and will execute that number of times or until  
 2 SUM exceeds SUM\_MAX, in which case the EXIT OUTER statement terminates both loops. The inner  
 3 loop is skipped by the first CYCLE statement if the quality flag, IQUAL, is too low. If CALCULATE  
 4 returns a negative RESULT, the second CYCLE statement prevents it from being summed. Both loops  
 5 have construct names and the inner loop also has a label. A construct name is required in the EXIT  
 6 statement in order to terminate both loops, but is optional in the CYCLE statements because each  
 7 belongs to its innermost loop.

8 Example 2:

```

9      N = 0
10     DO 50, I = 1, 10
11         J = I
12         DO K = 1, 5
13             L = K
14             N = N + 1 ! This statement executes 50 times
15         END DO      ! Nonlabeled DO inside a labeled DO
16     50 CONTINUE

```

17 After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

18 Example 3:

```

19      N = 0
20      DO I = 1, 10
21          J = I
22          DO 60, K = 5, 1 ! This inner loop is never executed
23              L = K
24              N = N + 1
25      60 CONTINUE      ! Labeled DO inside a nonlabeled DO
26      END DO

```

27 After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by  
 28 these statements.

29 The following are all valid examples of nonblock DO constructs:

30 Example 4:

```

31      DO 70
32          READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
33          IF (IOS /= 0) EXIT
34          IF (X < 0.) GOTO 70
35          CALL SUBA (X)
36          CALL SUBB (X)
37          ...
38          CALL SUBY (X)
39          CYCLE
40      70 CALL SUBNEG (X) ! SUBNEG called only when X < 0.

```

1 This is not a block DO construct because it ends with a statement other than END DO or CONTINUE. The loop will  
 2 continue to execute until an end-of-file condition or input/output error occurs.

3 Example 5:

```

4      SUM = 0.0
5      READ (IUN) N
6      DO 80, L = 1, N
7          READ (IUN) IQUAL, M, ARRAY (1:M)
8          IF (IQUAL < IQUAL_MIN) M = 0  ! Skip inner loop
9          DO 80 I = 1, M
10             CALL CALCULATE (ARRAY (I), RESULT)
11             IF (RESULT < 0.) CYCLE
12             SUM = SUM + RESULT
13             IF (SUM > SUM_MAX) GOTO 81
14      80  CONTINUE  ! This CONTINUE is shared by both loops
15      81 CONTINUE

```

16 This example is similar to Example 1 above, except that the two loops are not block DO constructs because they share  
 17 the CONTINUE statement with the label 80. The terminal construct of the outer DO is the entire inner DO construct.  
 18 The inner loop is skipped by forcing M to zero. If SUM grows too large, both loops are terminated by branching to the  
 19 CONTINUE statement labeled 81. The CYCLE statement in the inner loop is used to skip negative values of RESULT.

20 Example 6:

```

21      N = 0
22      DO 100 I = 1, 10
23          J = I
24          DO 100 K = 1, 5
25              L = K
26      100      N = N + 1  ! This statement executes 50 times

```

27 In this example, the two loops share an assignment statement. After execution of this program fragment, I = 11, J = 10,  
 28 K = 6, L = 5, and N = 50.

29 Example 7:

```

30      N = 0
31      DO 200 I = 1, 10
32          J = I
33          DO 200 K = 5, 1  ! This inner loop is never executed
34              L = K
35      200      N = N + 1

```

36 This example is very similar to the previous one, except that the inner loop is never executed. After execution of this  
 37 program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

### 38 C.5.4 Examples of invalid DO constructs

39 The following are all examples of invalid skeleton DO constructs:

40 Example 1:

```
1 DO I = 1, 10
2   ...
3 END DO LOOP    ! No matching construct name
```

4 Example 2:

```
5 LOOP: DO 1000 I = 1, 10    ! No matching construct name
6   ...
7 1000 CONTINUE
```

8 Example 3:

```
9 LOOP1: DO
10   ...
11 END DO LOOP2    ! Construct names don't match
```

12 Example 4:

```
13 DO I = 1, 10    ! Label required or ...
14   ...
15 1010 CONTINUE  ! ... END DO required
```

16 Example 5:

```
17 DO 1020 I = 1, 10
18   ...
19 1021 END DO      ! Labels don't match
```

20 Example 6:

```
21 FIRST: DO I = 1, 10
22   SECOND: DO J = 1, 5
23   ...
24   END DO FIRST    ! Improperly nested DOs
25 END DO SECOND
```

## 26 C.6 Clause 9 notes

### 27 C.6.1 External files (9.2)

28 This standard accommodates, but does not require, file cataloging. To do this, several concepts are  
29 introduced.

### 1 C.6.1.1 File connection (9.4)

2 Before any input/output may be performed on a file, it shall be connected to a unit. The unit then serves  
 3 as a designator for that file as long as it is connected. To be connected does not imply that “buffers”  
 4 have or have not been allocated, that “file-control tables” have or have not been filled, or that any other  
 5 method of implementation has been used. Connection means that (barring some other fault) a READ  
 6 or WRITE statement may be executed on the unit, hence on the file. Without a connection, a READ  
 7 or WRITE statement shall not be executed.

### 8 C.6.1.2 File existence (9.2.1)

9 Totally independent of the connection state is the property of existence, this being a file property. The  
 10 processor “knows” of a set of files that exist at a given time for a given program. This set would include  
 11 tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set may exclude files inaccessible  
 12 to the program because of security, because they are already in use by another program, etc. This  
 13 standard does not specify which files exist, hence wide latitude is available to a processor to implement  
 14 security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of the files  
 15 that a program can potentially process.

16 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

17 Means are provided to create, delete, connect, and disconnect files.

### 18 C.6.1.3 File names (9.4.5.8)

19 A file may have a name. The form of a file name is not specified. If a system does not have some form of  
 20 cataloging or tape labeling for at least some of its files, all file names will disappear at the termination  
 21 of execution. This is a valid implementation. Nowhere does this standard require names to survive for  
 22 any period of time longer than the execution time span of a program. Therefore, this standard does not  
 23 impose cataloging as a prerequisite. The naming feature is intended to allow use of a cataloging system  
 24 where one exists.

### 25 C.6.1.4 File access (9.2.2)

26 This standard does not address problems of security, protection, locking, and many other concepts that  
 27 may be part of the concept of “right of access”. Such concepts are considered to be in the province of  
 28 an operating system.

29 The OPEN and INQUIRE statements can be extended naturally to consider these things.

30 Possible access methods for a file are: sequential and direct. The processor may implement two different  
 31 types of files, each with its own access method. It might also implement one type of file with two different  
 32 access methods.

33 Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key  
 34 is a positive integer.

## 1 C.6.2 Nonadvancing input/output (9.2.3.1)

2 Data transfer statements affect the positioning of an external file. In FORTRAN 77, if no error or end-of-  
 3 file condition exists, the file is positioned after the record just read or written and that record becomes  
 4 the preceding record. This standard contains the record positioning ADVANCE= specifier in a data  
 5 transfer statement that provides the capability of maintaining a position within the current record from  
 6 one formatted data transfer statement to the next data transfer statement. The value NO provides this  
 7 capability. The value YES positions the file after the record just read or written. The default is YES.

8 The tab edit descriptor and the slash are still appropriate for use with this type of record access but the  
 9 tab will not reposition before the left tab limit.

10 A BACKSPACE of a file that is positioned within a record causes the specified unit to be positioned  
 11 before the current record.

12 If the last data transfer statement was WRITE and the file is positioned within a record, the file will be  
 13 positioned implicitly after the current record before an ENDFILE record is written to the file, that is, a  
 14 REWIND, BACKSPACE, or ENDFILE statement following a nonadvancing WRITE statement causes  
 15 the file to be positioned at the end of the current output record before the endfile record is written to  
 16 the file.

17 This standard provides a SIZE= specifier to be used with nonadvancing data transfer statements. The  
 18 variable in the SIZE= specifier will contain the count of the number of characters that make up the  
 19 sequence of values read by the data edit descriptors in this input statement.

20 The count is especially helpful if there is only one list item in the input list because it will contain the  
 21 number of characters that were present for the item.

22 The EOR= specifier is provided to indicate when an end-of-record condition has been encountered during  
 23 a nonadvancing data transfer statement. The end-of-record condition is not an error condition. If this  
 24 specifier is present, the current input list item that required more characters than the record contained  
 25 will be padded with blanks if PAD= 'YES' is in effect. This means that the input list item was successfully  
 26 completed. The file will then be positioned after the current record. The IOSTAT= specifier, if present,  
 27 will be defined with the value of the named constant IOSTAT\_EOR from the ISO\_FORTRAN\_ENV  
 28 module and the data transfer statement will be terminated. Program execution will continue with the  
 29 statement specified in the EOR= specifier. The EOR= specifier gives the capability of taking control  
 30 of execution when the end-of-record has been found. The *do-variables* in *io-implied-dos* retain their  
 31 last defined value and any remaining items in the *input-item-list* retain their definition status when an  
 32 end-of-record condition occurs. The SIZE= specifier, if present, will contain the number of characters  
 33 read with the data edit descriptors during this READ statement.

34 For nonadvancing input, the processor is not required to read partial records. The processor may read  
 35 the entire record into an internal buffer and make successive portions of the record available to successive  
 36 input statements.

37 In an implementation of nonadvancing input/output in which a nonadvancing write to a terminal device  
 38 causes immediate display of the output, such a write can be used as a mechanism to output a prompt.  
 39 In this case, the statement

```
40 WRITE (*, FMT='(A)', ADVANCE='NO') 'CONTINUE?(Y/N): '
```

41 would result in the prompt

```
42 CONTINUE?(Y/N):
```

43 being displayed with no subsequent line feed.

1 The response, which might be read by a statement of the form

2 `READ (*, FMT='(A)') ANSWER`

3 can then be entered on the same line as the prompt as in

4 `CONTINUE?(Y/N): Y`

5 The standard does not require that an implementation of nonadvancing input/output operate in this  
6 manner. For example, an implementation of nonadvancing output in which the display of the output is  
7 deferred until the current record is complete is also standard-conforming. Such an implementation will  
8 not, however, allow a prompting mechanism of this kind to operate.

### 9 **C.6.3 Asynchronous input/output**

10 Rather than limit support for asynchronous input/output to what has been traditionally provided by  
11 facilities such as `BUFFERIN/BUFFEROUT`, this standard builds upon existing Fortran syntax. This  
12 permits alternative approaches for implementing asynchronous input/output, and simplifies the task of  
13 adapting existing standard-conforming programs to use asynchronous input/output.

14 Not all processors will actually perform input/output asynchronously, nor will every processor that does  
15 be able to handle data transfer statements with complicated input/output item lists in an asynchronous  
16 manner. Such processors can still be standard-conforming. Hopefully, the documentation for each  
17 Fortran processor will describe when, if ever, input/output will be performed asynchronously.

18 This standard allows for at least two different conceptual models for asynchronous input/output.

19 Model 1: the processor will perform asynchronous input/output when the item list is simple (perhaps  
20 one contiguous named array) and the input/output is unformatted. The implementation cost is reduced,  
21 and this is the scenario most likely to be beneficial on traditional “big-iron” machines.

22 Model 2: The processor is free to do any of the following:

- 23 (1) on output, create a buffer inside the input/output library, completely formatted, and then  
24 start an asynchronous write of the buffer, and immediately return to the next statement in  
25 the program. The processor is free to wait for previously issued `WRITES`, or not, or
- 26 (2) pass the input/output list addresses to another processor/process, which will process the  
27 list items independently of the processor that executes the user’s code. The addresses of the  
28 list items must be computed before the asynchronous `READ/WRITE` statement completes.  
29 There is still an ordering requirement on list item processing to handle things like `READ`  
30 `(...) N,(a(i),i=1,N)`.

31 The standard allows a user to issue a large number of asynchronous input/output requests, without  
32 waiting for any of them to complete, and then wait for any or all of them. It may be impossible, and  
33 undesirable to keep track of each of these input/output requests individually.

34 It is not necessary for all requests to be tracked by the runtime library. If an `ID=` specifier does not  
35 appear in on a `READ` or `WRITE` statement, the runtime is free to forget about this particular request  
36 once it has successfully completed. If it gets an `ERR` or `END` condition, the processor is free to report  
37 this during any input/output operation to that unit. When an `ID=` specifier is present, the processor’s  
38 runtime input/output library is required to keep track of any `END` or `ERR` conditions for that particular  
39 input/output request. However, if the input/output request succeeds without any exceptional conditions  
40 occurring, then the runtime can forget that `ID=` value if it wishes. Typically, a runtime might only keep  
41 track of the last request made, or perhaps a very few. Then, when a user `WAITs` for a particular request,  
42 either the library knows about it (and does the right thing with respect to error handling, etc.), or will  
43 assume it is one of those requests that successfully completed and was forgotten about (and will just  
44 return without signaling any end or error conditions). It is incumbent on the user to pass valid `ID=`

values. There is no requirement on the processor to detect invalid ID= values. There is of course, a processor dependent limit on how many outstanding input/output requests that generate an end or error condition can be handled before the processor runs out of memory to keep track of such conditions. The restrictions on the SIZE= variables are designed to allow the processor to update such variables at any time (after the request has been processed, but before the WAIT operation), and then forget about them. That's why there is no SIZE= specifier allowed in the various WAIT operations. Only exceptional conditions (errors or ends of files) are expected to be tracked by individual request by the runtime, and then only if an ID= specifier was present. The END= and EOR= specifiers have not been added to all statements that can be WAIT operations. Instead, the IOSTAT variable will have to be queried after a WAIT operation to handle this situation. This choice was made because we expect the WAIT statement to be the usual method of waiting for input/output to complete (and WAIT does support the END= and EOR= specifiers). This particular choice is philosophical, and was not based on significant technical difficulties.

Note that the requirement to set the IOSTAT variable correctly requires an implementation to remember which input/output requests got an EOR condition, so that a subsequent wait operation will return the correct IOSTAT value. This means there is a processor defined limit on the number of outstanding nonadvancing input/output requests that got an EOR condition (constrained by available memory to keep track of this information, similar to END/ERR conditions).

#### C.6.4 OPEN statement (9.4.5)

A file may become connected to a unit either by preconnection or by execution of an OPEN statement. Preconnection is performed prior to the beginning of execution of a program by means external to Fortran. For example, it may be done by job control action or by processor-established defaults. Execution of an OPEN statement is not required to access preconnected files (9.4.4).

The OPEN statement provides a means to access existing files that are not preconnected. An OPEN statement may be used in either of two ways: with a file name (open-by-name) and without a file name (open-by-unit). A unit is given in either case. Open-by-name connects the specified file to the specified unit. Open-by-unit connects a processor-dependent default file to the specified unit. (The default file might or might not have a name.)

Therefore, there are three ways a file may become connected and hence processed: preconnection, open-by-name, and open-by-unit. Once a file is connected, there is no means in standard Fortran to determine how it became connected.

An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection methods may be performed on a file that does not exist. When a unit is preconnected, writing the first record creates the file. With the other two methods, execution of the OPEN statement creates the file.

When an OPEN statement is executed, the unit specified in the OPEN might or might not already be connected to a file. If it is already connected to a file (either through preconnection or by a prior OPEN), then omitting the FILE= specifier in the OPEN statement implies that the file is to remain connected to the unit. Such an OPEN statement may be used to change the values of the blank interpretation mode, decimal edit mode, pad mode, input/output rounding mode, delimiter mode, and sign mode.

If the value of the ACTION= specifier is WRITE, then 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.

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.



```

1 CHARACTER (LEN = 20) CH1
2 WRITE (10, '(A)') 'THIS IS RECORD 1'
3 OPEN (UNIT = 10, STATUS = 'OLD', PAD = 'YES')
4 REWIND 10
5 READ (10, '(A20)') CH1    ! CH1 now has the value
6                          ! 'THIS IS RECORD 1    '

```

7 In the next example, unit 12 is first connected to a file named FRED, with a status of OLD. The second  
 8 OPEN statement then opens unit 12 again, retaining the connection to the file FRED, but changing the  
 9 value of the DELIM= specifier to QUOTE.

```

10 CHARACTER (LEN = 25) CH2, CH3
11 OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
12 CH2 = '''THIS STRING HAS QUOTES.'''
13      ! Quotes in string CH2
14 WRITE (12, *) CH2          ! Written with no delimiters
15 OPEN (12, DELIM = 'QUOTE') ! Now quote is the delimiter
16 REWIND 12
17 READ (12, *) CH3    ! CH3 now has the value
18                  ! 'THIS STRING HAS QUOTES.  '

```

19 The next example is invalid because it attempts to change the value of the STATUS= specifier.

```

20 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
21 WRITE (10, *) A, B, C
22 OPEN (10, STATUS = 'SCRATCH')    ! Attempts to make FRED
23                               ! a SCRATCH file

```

24 The previous example could be made valid by closing the unit first, as in the next example.

```

25 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
26 WRITE (10, *) A, B, C
27 CLOSE (10)
28 OPEN (10, STATUS = 'SCRATCH')    ! Opens a different
29                               ! SCRATCH file

```

### 30 C.6.5 Connection properties (9.4.3)

31 When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection,  
 32 the following connection properties, among others, may be established.

- 33 (1) An access method, which is sequential, direct, or stream, is established for the connection  
 34 (9.4.5.1).
- 35 (2) A form, which is formatted or unformatted, is established for a connection to a file that  
 36 exists or is created by the connection. For a connection that results from execution of an  
 37 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 (9.4.6.3).

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 standard 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 (9.4.7)

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, when a format rescan occurs at the end of the format, and at completion of execution of the output statement (even if the output list is empty). Thus, the occurrence of  $n$  successive slashes between two other edit descriptors causes  $n - 1$  blank lines if the records are printed. The occurrence of  $n$  slashes at the beginning or end of a complete format specification causes  $n$  blank lines if the records are printed. However, a complete format specification containing  $n$  slashes ( $n > 0$ ) and no other edit descriptors causes  $n + 1$  blank lines if the records are printed. For example, the statements

```
PRINT 3
3 FORMAT (/)
```

1 will write two records that cause two blank lines if the records are printed.

## 2 **C.7.2 List-directed input (10.10.1)**

3 The following examples illustrate list-directed input. A blank character is represented by b.

4 Example 1:

5 Program:

```
6 J = 3
7 READ *, I
8 READ *, J
```

9 Sequential input file:

```
10 record 1:  b1b,4bbbbbb
11 record 2:  ,2bbbbbbbbb
```

12 Result: I = 1, J = 3.

13 Explanation: The second READ statement reads the second record. The initial comma in the record  
14 designates a null value; therefore, J is not redefined.

15 Example 2:

16 Program:

```
17 CHARACTER A *8, B *1
18 READ *, A, B
```

19 Sequential input file:

```
20 record 1:  'bbbbbbbbb'
21 record 2:  'QXY'b'Z'
```

22 Result: A = 'bbbbbbbbb', B = 'Q'

23 Explanation: In the first record, the rightmost apostrophe is interpreted as delimiting the constant (it  
24 cannot be the first of a pair of embedded apostrophes representing a single apostrophe because this  
25 would involve the prohibited “splitting” of the pair by the end of a record); therefore, A is assigned  
26 the character constant 'bbbbbbbbb'. The end of a record acts as a blank, which in this case is a value  
27 separator because it occurs between two constants.

## 28 **C.8 Clause 11 notes**

### 29 **C.8.1 Main program and block data program unit (11.1, 11.3)**

30 The name of the main program or of a block data program unit has no explicit use within the Fortran  
31 language. It is available for documentation and for possible use by a processor.

1 A processor may implement an unnamed main program or unnamed block data program unit by assigning  
2 it a default name. However, this name shall not conflict with any other global name in a standard-  
3 conforming program. This might be done by making the default name one that is not permitted in a  
4 standard-conforming program (for example, by including a character not normally allowed in names)  
5 or by providing some external mechanism such that for any given program the default name can be  
6 changed to one that is otherwise unused.

## 7 **C.8.2 Dependent compilation (11.2)**

8 This standard, like its predecessors, is intended to permit the implementation of conforming processors  
9 in which a program can be broken into multiple units, each of which can be separately translated in  
10 preparation for execution. Such processors are commonly described as supporting separate compilation.  
11 There is an important difference between the way separate compilation can be implemented under this  
12 standard and the way it could be implemented under the FORTRAN 77 International Standard. Under  
13 the FORTRAN 77 standard, any information required to translate a program unit was specified in that  
14 program unit. Each translation was thus totally independent of all others. Under this standard, a  
15 program unit can use information that was specified in a separate module and thus may be dependent  
16 on that module. The implementation of this dependency in a processor may be that the translation of a  
17 program unit may depend on the results of translating one or more modules. Processors implementing  
18 the dependency this way are commonly described as supporting dependent compilation.

19 The dependencies involved here are new only in the sense that the Fortran processor is now aware of  
20 them. The same information dependencies existed under the FORTRAN 77 International Standard, but  
21 it was the programmer's responsibility to transport the information necessary to resolve them by making  
22 redundant specifications of the information in multiple program units. The availability of separate but  
23 dependent compilation offers several potential advantages over the redundant textual specification of  
24 information.

- 25 (1) Specifying information at a single place in the program ensures that different program units  
26 using that information will be translated consistently. Redundant specification leaves the  
27 possibility that different information will erroneously be specified. Even if an INCLUDE line  
28 is used to ensure that the text of the specifications is identical in all involved program units,  
29 the presence of other specifications (for example, an IMPLICIT statement) may change the  
30 interpretation of that text.
- 31 (2) During the revision of a program, it is possible for a processor to assist in determining  
32 whether different program units have been translated using different (incompatible) versions  
33 of a module, although there is no requirement that a processor provide such assistance.  
34 Inconsistencies in redundant textual specification of information, on the other hand, tend  
35 to be much more difficult to detect.
- 36 (3) Putting information in a module provides a way of packaging it. Without modules, redun-  
37 dant specifications frequently shall be interleaved with other specifications in a program  
38 unit, making convenient packaging of such information difficult.
- 39 (4) Because a processor may be implemented such that the specifications in a module are  
40 translated once and then repeatedly referenced, there is the potential for greater efficiency  
41 than when the processor shall translate redundant specifications of information in multiple  
42 program units.

43 The exact meaning of the requirement that the public portions of a module be available at the time of  
44 reference is processor dependent. For example, a processor could consider a module to be available only  
45 after it has been compiled and require that if the module has been compiled separately, the result of  
46 that compilation shall be identified to the compiler when compiling program units that use it.

### 1 C.8.2.1 USE statement and dependent compilation (11.2.1)

2 Another benefit of the USE statement is its enhanced facilities for name management. If one needs to  
3 use only selected entities in a module, one can do so without having to worry about the names of all  
4 the other entities in that module. If one needs to use two different modules that happen to contain  
5 entities with the same name, there are several ways to deal with the conflict. If none of the entities with  
6 the same name are to be used, they can simply be ignored. If the name happens to refer to the same  
7 entity in both modules (for example, if both modules obtained it from a third module), then there is no  
8 confusion about what the name denotes and the name can be freely used. If the entities are different  
9 and one or both is to be used, the local renaming facility in the USE statement makes it possible to give  
10 those entities different names in the program unit containing the USE statements.

11 A benefit of using the ONLY option consistently, as compared to USE without it, is that the module  
12 from which each accessed entity is accessed is explicitly specified in each program unit. This means that  
13 one need not search other program units to find where each one is defined. This reduces maintenance  
14 costs.

15 A typical implementation of dependent but separate compilation may involve storing the result of trans-  
16 lating a module in a file (or file element) whose name is derived from the name of the module. Note,  
17 however, that the name of a module is limited only by the Fortran rules and not by the names allowed  
18 in the file system. Thus the processor may have to provide a mapping between Fortran names and file  
19 system names.

20 The result of translating a module could reasonably either contain only the information textually specified  
21 in the module (with “pointers” to information originally textually specified in other modules) or contain  
22 all information specified in the module (including copies of information originally specified in other  
23 modules). Although the former approach would appear to save on storage space, the latter approach  
24 can greatly simplify the logic necessary to process a USE statement and can avoid the necessity of  
25 imposing a limit on the logical “nesting” of modules via the USE statement.

26 Variables declared in a module retain their definition status on much the same basis as variables in  
27 a common block. That is, saved variables retain their definition status throughout the execution of a  
28 program, while variables that are not saved retain their definition status only during the execution of  
29 scoping units that reference the module. In some cases, it may be appropriate to put a USE statement  
30 such as

31 `USE MY_MODULE, ONLY:`

32 in a scoping unit in order to assure that other procedures that it references can communicate through  
33 the module. In such a case, the scoping unit would not access any entities from the module, but the  
34 variables not saved in the module would retain their definition status throughout the execution of the  
35 scoping unit.

36 There is an increased potential for undetected errors in a scoping unit that uses both implicit typing  
37 and the USE statement. For example, in the program fragment

```
38 SUBROUTINE SUB
39   USE MY_MODULE
40   IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
41   X = F (B)
42   A = G (X) + H (X + 1)
43 END SUBROUTINE SUB
```

44 X could be either an implicitly typed real variable or a variable obtained from the module MY\_MODULE

1 and might change from one to the other because of changes in MY\_MODULE unrelated to the action  
2 performed by SUB. Logic errors resulting from this kind of situation can be extremely difficult to locate.  
3 Thus, the use of these features together is discouraged.

#### 4 **C.8.2.2 Accessibility attributes**

5 The PUBLIC and PRIVATE attributes, which can be declared only in modules, divide the entities in a  
6 module into those that are actually relevant to a scoping unit referencing the module and those that are  
7 not. This information may be used to improve the performance of a Fortran processor. For example,  
8 it may be possible to discard much of the information about the private entities once a module has  
9 been translated, thus saving on both storage and the time to search it. Similarly, it may be possible  
10 to recognize that two versions of a module differ only in the private entities they contain and avoid  
11 retranslating program units that use that module when switching from one version of the module to the  
12 other.

### 13 **C.8.3 Examples of the use of modules**

#### 14 **C.8.3.1 Identical common blocks**

15 A common block and all its associated specification statements may be placed in a module named, for  
16 example, MY\_COMMON and accessed by a USE statement of the form

17 `USE MY_COMMON`

18 that accesses the whole module without any renaming. This ensures that all instances of the common  
19 block are identical. Module MY\_COMMON could contain more than one common block.

#### 20 **C.8.3.2 Global data**

21 A module may contain only data objects, for example:

```
22 MODULE DATA_MODULE
23   SAVE
24   REAL A (10), B, C (20,20)
25   INTEGER :: I=0
26   INTEGER, PARAMETER :: J=10
27   COMPLEX D (J,J)
28 END MODULE DATA_MODULE
```

29 Data objects made global in this manner may have any combination of data types.

30 Access to some of these may be made by a USE statement with the ONLY option, such as:

31 `USE DATA_MODULE, ONLY: A, B, D`

32 and access to all of them may be made by the following USE statement:

33 `USE DATA_MODULE`

34 Access to all of them with some renaming to avoid name conflicts may be made by:

35 `USE DATA_MODULE, AMODULE => A, DMODULE => D`

### 1 C.8.3.3 Derived types

2 A derived type may be defined in a module and accessed in a number of program units. For example:

```
3 MODULE SPARSE
4     TYPE NONZERO
5         REAL A
6         INTEGER I, J
7     END TYPE NONZERO
8 END MODULE SPARSE
```

9 defines a type consisting of a real component and two integer components for holding the numerical  
10 value of a nonzero matrix element and its row and column indices.

### 11 C.8.3.4 Global allocatable arrays

12 Many programs need large global allocatable arrays whose sizes are not known before program execution.  
13 A simple form for such a program is:

```
14 PROGRAM GLOBAL_WORK
15     CALL CONFIGURE_ARRAYS      ! Perform the appropriate allocations
16     CALL COMPUTE               ! Use the arrays in computations
17 END PROGRAM GLOBAL_WORK
18 MODULE WORK_ARRAYS            ! An example set of work arrays
19     INTEGER N
20     REAL, ALLOCATABLE, SAVE :: A (:), B (:, :), C (:, :, :)
21 END MODULE WORK_ARRAYS
22 SUBROUTINE CONFIGURE_ARRAYS    ! Process to set up work arrays
23     USE WORK_ARRAYS
24     READ (*, *) N
25     ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
26 END SUBROUTINE CONFIGURE_ARRAYS
27 SUBROUTINE COMPUTE
28     USE WORK_ARRAYS
29     ... ! Computations involving arrays A, B, and C
30 END SUBROUTINE COMPUTE
```

31 Typically, many subprograms need access to the work arrays, and all such subprograms would contain  
32 the statement

```
33 USE WORK_ARRAYS
```

### 34 C.8.3.5 Procedure libraries

35 Interface bodies for external procedures in a library may be gathered into a module. This permits the  
36 use of argument keywords and optional arguments, and allows static checking of the references. Different  
37 versions may be constructed for different applications, using argument keywords in common use in each  
38 application.

1 An example is the following library module:

```

2 MODULE LIBRARY_LLS
3   INTERFACE
4     SUBROUTINE LLS (X, A, F, FLAG)
5       REAL X (:, :)
6       ! The SIZE in the next statement is an intrinsic function
7       REAL, DIMENSION (SIZE (X, 2)) :: A, F
8       INTEGER FLAG
9     END SUBROUTINE LLS
10    ...
11  END INTERFACE
12    ...
13 END MODULE LIBRARY_LLS

```

14 This module allows the subroutine LLS to be invoked:

```

15 USE LIBRARY_LLS
16   ...
17 CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
18   ...

```

### 19 C.8.3.6 Operator extensions

20 In order to extend an intrinsic operator symbol to have an additional meaning, an interface block  
 21 specifying that operator symbol in the OPERATOR option of the INTERFACE statement may be  
 22 placed in a module.

23 For example, // may be extended to perform concatenation of two derived-type objects serving as varying  
 24 length character strings and + may be extended to specify matrix addition for type MATRIX or interval  
 25 arithmetic addition for type INTERVAL.

26 A module might contain several such interface blocks. An operator may be defined by an external  
 27 function (either in Fortran or some other language) and its procedure interface placed in the module.

### 28 C.8.3.7 Data abstraction

29 In addition to providing a portable means of avoiding the redundant specification of information in  
 30 multiple program units, a module provides a convenient means of “packaging” related entities, such as  
 31 the definitions of the representation and operations of an abstract data type. The following example  
 32 of a module defines a data abstraction for a SET type where the elements of each set are of type  
 33 integer. The standard set operations of UNION, INTERSECTION, and DIFFERENCE are provided.  
 34 The CARDINALITY function returns the cardinality of (number of elements in) its set argument.  
 35 Two functions returning logical values are included, ELEMENT and SUBSET. ELEMENT defines the  
 36 operator .IN. and SUBSET extends the operator <=. ELEMENT determines if a given scalar integer  
 37 value is an element of a given set, and SUBSET determines if a given set is a subset of another given  
 38 set. (Two sets may be checked for equality by comparing cardinality and checking that one is a subset  
 39 of the other, or checking to see if each is a subset of the other.)



1 The transfer function SETF converts a vector of integer values to the corresponding set, with duplicate  
2 values removed. Thus, a vector of constant values can be used as set constants. An inverse transfer  
3 function VECTOR returns the elements of a set as a vector of values in ascending order. In this SET  
4 implementation, set data objects have a maximum cardinality of 200.

```
5 MODULE INTEGER_SETS
6 ! This module is intended to illustrate use of the module facility
7 ! to define a new type, along with suitable operators.
8
9 INTEGER, PARAMETER :: MAX_SET_CARD = 200
10
11 TYPE SET                                ! Define SET type
12     PRIVATE
13     INTEGER CARD
14     INTEGER ELEMENT (MAX_SET_CARD)
15 END TYPE SET
16
17 INTERFACE OPERATOR (.IN.)
18     MODULE PROCEDURE ELEMENT
19 END INTERFACE OPERATOR (.IN.)
20
21 INTERFACE OPERATOR (<=)
22     MODULE PROCEDURE SUBSET
23 END INTERFACE OPERATOR (<=)
24
25 INTERFACE OPERATOR (+)
26     MODULE PROCEDURE UNION
27 END INTERFACE OPERATOR (+)
28
29 INTERFACE OPERATOR (-)
30     MODULE PROCEDURE DIFFERENCE
31 END INTERFACE OPERATOR (-)
32
33 INTERFACE OPERATOR (*)
34     MODULE PROCEDURE INTERSECTION
35 END INTERFACE OPERATOR (*)
36
37 CONTAINS
38
39 INTEGER FUNCTION CARDINALITY (A)        ! Returns cardinality of set A
40     TYPE (SET), INTENT (IN) :: A
41     CARDINALITY = A % CARD
42 END FUNCTION CARDINALITY
43
44 LOGICAL FUNCTION ELEMENT (X, A)         ! Determines if
```

```

1    INTEGER, INTENT(IN) :: X                ! element X is in set A
2    TYPE (SET), INTENT(IN) :: A
3    ELEMENT = ANY (A % ELEMENT (1 : A % CARD) == X)
4    END FUNCTION ELEMENT
5
6    FUNCTION UNION (A, B)                    ! Union of sets A and B
7    TYPE (SET) UNION
8    TYPE (SET), INTENT(IN) :: A, B
9    INTEGER J
10   UNION = A
11   DO J = 1, B % CARD
12       IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
13           IF (UNION % CARD < MAX_SET_CARD) THEN
14               UNION % CARD = UNION % CARD + 1
15               UNION % ELEMENT (UNION % CARD) = &
16                   B % ELEMENT (J)
17           ELSE
18               ! Maximum set size exceeded . . .
19           END IF
20       END IF
21   END DO
22 END FUNCTION UNION
23
24 FUNCTION DIFFERENCE (A, B)                 ! Difference of sets A and B
25 TYPE (SET) DIFFERENCE
26 TYPE (SET), INTENT(IN) :: A, B
27 INTEGER J, X
28 DIFFERENCE % CARD = 0                     ! The empty set
29 DO J = 1, A % CARD
30     X = A % ELEMENT (J)
31     IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, X)
32 END DO
33 END FUNCTION DIFFERENCE
34
35 FUNCTION INTERSECTION (A, B)               ! Intersection of sets A and B
36 TYPE (SET) INTERSECTION
37 TYPE (SET), INTENT(IN) :: A, B
38 INTERSECTION = A - (A - B)
39 END FUNCTION INTERSECTION
40
41 LOGICAL FUNCTION SUBSET (A, B)              ! Determines if set A is
42 TYPE (SET), INTENT(IN) :: A, B            ! a subset of set B
43 INTEGER I
44 SUBSET = A % CARD <= B % CARD
45 IF (.NOT. SUBSET) RETURN                  ! For efficiency

```

```

1      DO I = 1, A % CARD
2          SUBSET = SUBSET .AND. (A % ELEMENT (I) .IN. B)
3      END DO
4  END FUNCTION SUBSET
5
6  TYPE (SET) FUNCTION SETF (V)      ! Transfer function between a vector
7      INTEGER V (:)                ! of elements and a set of elements
8      INTEGER J                    ! removing duplicate elements
9      SETF % CARD = 0
10     DO J = 1, SIZE (V)
11         IF (.NOT. (V (J) .IN. SETF)) THEN
12             IF (SETF % CARD < MAX_SET_CARD) THEN
13                 SETF % CARD = SETF % CARD + 1
14                 SETF % ELEMENT (SETF % CARD) = V (J)
15             ELSE
16                 ! Maximum set size exceeded . . .
17             END IF
18         END IF
19     END DO
20 END FUNCTION SETF
21
22 FUNCTION VECTOR (A)                ! Transfer the values of set A
23     TYPE (SET), INTENT (IN) :: A    ! into a vector in ascending order
24     INTEGER, POINTER :: VECTOR (:)
25     INTEGER I, J, K
26     ALLOCATE (VECTOR (A % CARD))
27     VECTOR = A % ELEMENT (1 : A % CARD)
28     DO I = 1, A % CARD - 1          ! Use a better sort if
29         DO J = I + 1, A % CARD      ! A % CARD is large
30             IF (VECTOR (I) > VECTOR (J)) THEN
31                 K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
32             END IF
33         END DO
34     END DO
35 END FUNCTION VECTOR
36
37 END MODULE INTEGER_SETS

```

38 Examples of using INTEGER\_SETS (A, B, and C are variables of type SET; X  
39 is an integer variable):  
40 ! Check to see if A has more than 10 elements  
41 IF (CARDINALITY (A) > 10) ...  
42  
43 ! Check for X an element of A but not of B  
44 IF (X .IN. (A - B)) ...

```

1
2 ! C is the union of A and the result of B intersected
3 ! with the integers 1 to 100
4 C = A + B * SETF ((/ (I, I = 1, 100) /))
5
6 ! Does A have any even numbers in the range 1:100?
7 IF (CARDINALITY (A * SETF ((/ (I, I = 2, 100, 2) /))) > 0) ...
8
9 PRINT *, VECTOR (B) ! Print out the elements of set B, in ascending order

```

### 10 C.8.3.8 Public entities renamed

11 At times it may be necessary to rename entities that are accessed with USE statements. Care should be  
 12 taken if the referenced modules also contain USE statements.

13 The following example illustrates renaming features of the USE statement.

```

14 MODULE J; REAL JX, JY, JZ; END MODULE J
15 MODULE K
16     USE J, ONLY : KX => JX, KY => JY
17     ! KX and KY are local names to module K
18     REAL KZ      ! KZ is local name to module K
19     REAL JZ      ! JZ is local name to module K
20 END MODULE K
21 PROGRAM RENAME
22     USE J; USE K
23     ! Module J's entity JX is accessible under names JX and KX
24     ! Module J's entity JY is accessible under names JY and KY
25     ! Module K's entity KZ is accessible under name KZ
26     ! Module J's entity JZ and K's entity JZ are different entities
27     ! and shall not be referenced
28     ...
29 END PROGRAM RENAME

```

### 30 C.8.4 Modules with submodules

31 Each submodule specifies that it is the child of exactly one parent module or submodule. Therefore, a  
 32 module and all of its descendant submodules stand in a tree-like relationship one to another.

33 If a module procedure interface body that is specified in a module has public accessibility, and its  
 34 corresponding separate module procedure is defined in a descendant of that module, the procedure can  
 35 be accessed by use association. No other entity in a submodule can be accessed by use association. Each  
 36 program unit that references a module by use association depends on it, and each submodule depends  
 37 on its ancestor module. Therefore, if one changes a separate module procedure body in a submodule but  
 38 does not change its corresponding module procedure interface, a tool for automatic program translation  
 39 would not need to reprocess program units that reference the module by use association. This is so  
 40 even if the tool exploits the relative modification times of files as opposed to comparing the result of  
 41 translating the module to the result of a previous translation.

1 By constructing taller trees, one can put entities at intermediate levels that are shared by submodules  
 2 at lower levels; changing these entities cannot change the interpretation of anything that is accessible  
 3 from the module by use association. Developers of modules that embody large complicated concepts  
 4 can exploit this possibility to organize components of the concept into submodules, while preserving the  
 5 privacy of entities that are shared by the submodules and that ought not to be exposed to users of the  
 6 module. Putting these shared entities at an intermediate level also prevents cascades of reprocessing  
 7 and testing if some of them are changed.

8 The following example illustrates a module, `color_points`, with a submodule, `color_points.a`, that in  
 9 turn has a submodule, `color_points.b`. Public entities declared within `color_points` can be accessed by  
 10 use association. The submodules `color_points.a` and `color_points.b` can be changed without causing  
 11 retranslation of program units that reference the module `color_points`.

12 The module `color_points` does not have a *module-subprogram-part*, but a *module-subprogram-part* is not  
 13 prohibited. The module could be published as definitive specification of the interface, without revealing  
 14 trade secrets contained within `color_points.a` or `color_points.b`. Of course, a similar module without  
 15 the `module` prefix in the interface bodies would serve equally well as documentation – but the procedures  
 16 would be external procedures. It would make little difference to the consumer, but the developer would  
 17 forfeit all of the advantages of modules.

```

18  module color_points
19
20      type color_point
21          private
22          real :: x, y
23          integer :: color
24      end type color_point
25
26      interface                ! Interfaces for procedures with separate
27                              ! bodies in the submodule color_points.a
28          module subroutine color_point_del ( p ) ! Destroy a color_point object
29              type(color_point), allocatable :: p
30          end subroutine color_point_del
31          ! Distance between two color_point objects
32          real module function color_point_dist ( a, b )
33              type(color_point), intent(in) :: a, b
34          end function color_point_dist
35          module subroutine color_point_draw ( p ) ! Draw a color_point object
36              type(color_point), intent(in) :: p
37          end subroutine color_point_draw
38          module subroutine color_point_new ( p ) ! Create a color_point object
39              type(color_point), allocatable :: p
40          end subroutine color_point_new
41      end interface
42
43  end module color_points
  
```

44 The only entities within `color_points.a` that can be accessed by use association are separate module  
 45 procedures for which corresponding module procedure interface bodies are provided in `color_points`.

1 If the procedures are changed but their interfaces are not, the interface from program units that access  
 2 them by use association is unchanged. If the module and submodule are in separate files, utilities that  
 3 examine the time of modification of a file would notice that changes in the module could affect the  
 4 translation of its submodules or of program units that reference the module by use association, but  
 5 that changes in submodules could not affect the translation of the parent module or program units that  
 6 reference it by use association.

7 The variable `instance_count` is not accessible by use association of `color_points`, but is accessible  
 8 within `color_points_a`, and its submodules.

```

9  submodule ( color_points ) color_points_a ! Submodule of color_points
10
11  integer, save :: instance_count = 0
12
13  interface                                ! Interface for a procedure with a separate
14                                          ! body in submodule color_points_b
15  module subroutine inquire_palette ( pt, pal )
16      use palette_stuff                    ! palette_stuff, especially submodules
17                                          ! thereof, can reference color_points by use
18                                          ! association without causing a circular
19                                          ! dependence during translation because this
20                                          ! use is not in the module. Furthermore,
21                                          ! changes in the module palette_stuff do not
22                                          ! affect the translation of color_points.
23      type(color_point), intent(in) :: pt
24      type(palette), intent(out) :: pal
25  end subroutine inquire_palette
26
27  end interface
28
29  contains ! Invisible bodies for public module procedure interfaces
30      ! declared in the module
31
32  module subroutine color_point_del ( p )
33      type(color_point), allocatable :: p
34      instance_count = instance_count - 1
35      deallocate ( p )
36  end subroutine color_point_del
37  real module function color_point_dist ( a, b ) result ( dist )
38      type(color_point), intent(in) :: a, b
39      dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
40  end function color_point_dist
41  module subroutine color_point_new ( p )
42      type(color_point), allocatable :: p
43      instance_count = instance_count + 1
44      allocate ( p )

```

```

1      end subroutine color_point_new
2
3  end submodule color_points_a

```

4 The subroutine `inquire_palette` is accessible within `color_points_a` because its interface is declared  
5 therein. It is not, however, accessible by use association, because its interface is not declared in the  
6 module, `color_points`. Since the interface is not declared in the module, changes in the interface  
7 cannot affect the translation of program units that reference the module by use association.

```

8  submodule ( color_points:color_points_a ) color_points_b ! Subsidiary**2 submodule
9
10 contains
11     ! Invisible body for interface declared in the ancestor module
12     module subroutine color_point_draw ( p )
13         use palette_stuff, only: palette
14         type(color_point), intent(in) :: p
15         type(palette) :: MyPalette
16         ...; call inquire_palette ( p, MyPalette ); ...
17     end subroutine color_point_draw
18
19     ! Invisible body for interface declared in the parent submodule
20     module procedure inquire_palette
21         ... implementation of inquire_palette
22     end procedure inquire_palette
23
24     subroutine private_stuff ! not accessible from color_points_a
25         ...
26     end subroutine private_stuff
27
28 end submodule color_points_b
29
30 module palette_stuff
31     type :: palette ; ... ; end type palette
32 contains
33     subroutine test_palette ( p )
34         ! Draw a color wheel using procedures from the color_points module
35         type(palette), intent(in) :: p
36         use color_points ! This does not cause a circular dependency because
37                         ! the "use palette_stuff" that is logically within
38                         ! color_points is in the color_points_a submodule.
39         ...
40     end subroutine test_palette
41 end module palette_stuff

```

42 There is a `use palette_stuff` in `color_points_a`, and a `use color_points` in `palette_stuff`. The  
43 `use palette_stuff` would cause a circular reference if it appeared in `color_points`. In this case, it does

1 not cause a circular dependence because it is in a submodule. Submodules cannot be referenced by use  
 2 association, and therefore what would be a circular appearance of use `palette_stuff` is not accessed.

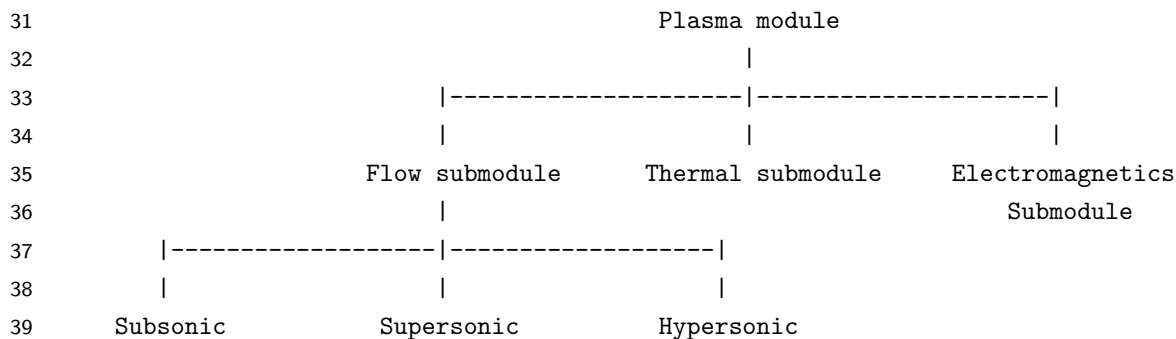
```

3  program main
4      use color_points
5      ! "instance_count" and "inquire_palette" are not accessible here
6      ! because they are not declared in the "color_points" module.
7      ! "color_points_a" and "color_points_b" cannot be referenced by
8      ! use association.
9      interface draw                      ! just to demonstrate it's possible
10         module procedure color_point_draw
11     end interface
12     type(color_point) :: C_1, C_2
13     real :: RC
14     ...
15     call color_point_new (c_1)          ! body in color_points_a, interface in color_points
16     ...
17     call draw (c_1)                    ! body in color_points_b, specific interface
18                                     ! in color_points, generic interface here.
19     ...
20     rc = color_point_dist (c_1, c_2) ! body in color_points_a, interface in color_points
21     ...
22     call color_point_del (c_1)         ! body in color_points_a, interface in color_points
23     ...
24 end program main

```

25 A multilevel submodule system can be used to package and organize a large and interconnected concept  
 26 without exposing entities of one subsystem to other subsystems.

27 Consider a **Plasma** module from a Tokamak simulator. A plasma simulation requires attention at least to  
 28 fluid flow, thermodynamics, and electromagnetism. Fluid flow simulation requires simulation of subsonic,  
 29 supersonic, and hypersonic flow. This problem decomposition can be reflected in the submodule structure  
 30 of the **Plasma** module:



40 Entities can be shared among the **Subsonic**, **Supersonic**, and **Hypersonic** submodules by putting  
 41 them within the **Flow** submodule. One then need not worry about accidental use of these entities by  
 42 use association or by the **Thermal** or **Electromagnetics** modules, or the development of a dependency  
 43 of correct operation of those subsystems upon the representation of entities of the **Flow** subsystem as a



1 consequence of maintenance. Since these these entities are not accessible by use association, if any of  
2 them are changed, the new values cannot be accessed in program units that reference the **Plasma** module  
3 by use association; the answer to the question “where are these entities used” is therefore confined to  
4 the set of descendant submodules of the **Flow** submodule.

## 5 **C.9 Clause 12 notes**

### 6 **C.9.1 Portability problems with external procedures (12.4.2.2)**

7 There is a potential portability problem in a scoping unit that references an external procedure without  
8 explicitly declaring it to have the **EXTERNAL** attribute (5.3.8). On a different processor, the name  
9 of that procedure may be the name of a nonstandard intrinsic procedure and the processor would  
10 be permitted to interpret those procedure references as references to that intrinsic procedure. (On that  
11 processor, the program would also be viewed as not conforming to the standard because of the references  
12 to the nonstandard intrinsic procedure.) Declaration of the **EXTERNAL** attribute causes the references  
13 to be to the external procedure regardless of the availability of an intrinsic procedure with the same  
14 name. Note that declaration of the type of a procedure is not enough to make it external, even if the  
15 type is inconsistent with the type of the result of an intrinsic of the same name.

### 16 **C.9.2 Procedures defined by means other than Fortran (12.6.3)**

17 A processor is not required to provide any means other than Fortran for defining external procedures.  
18 Among the means that might be supported are the machine assembly language, other high level lan-  
19 guages, the Fortran language extended with nonstandard features, and the Fortran language as supported  
20 by another Fortran processor (for example, a previously existing **FORTRAN 77** processor).

21 Procedures defined by means other than Fortran are considered external procedures because their def-  
22 initions are not in a Fortran program unit and because they are referenced using global names. The  
23 use of the term external should not be construed as any kind of restriction on the way in which these  
24 procedures may be defined. For example, if the means other than Fortran has its own facilities for  
25 internal and external procedures, it is permissible to use them. If the means other than Fortran can  
26 create an “internal” procedure with a global name, it is permissible for such an “internal” procedure  
27 to be considered by Fortran to be an external procedure. The means other than Fortran for defining  
28 external procedures, including any restrictions on the structure for organization of those procedures, are  
29 entirely processor dependent.

30 A Fortran processor may limit its support of procedures defined by means other than Fortran such that  
31 these procedures may affect entities in the Fortran environment only on the same basis as procedures  
32 written in Fortran. For example, it might prohibit the value of a local variable from being changed by  
33 a procedure reference unless that variable were one of the arguments to the procedure.

### 34 **C.9.3 Procedure interfaces (12.4)**

35 In **FORTRAN 77**, the interface to an external procedure was always deduced from the form of references  
36 to that procedure and any declarations of the procedure name in the referencing program unit. In this  
37 standard, features such as argument keywords and optional arguments make it impossible to deduce  
38 sufficient information about the dummy arguments from the nature of the actual arguments to be  
39 associated with them, and features such as array function results and pointer function results make  
40 necessary extensions to the declaration of a procedure that cannot be done in a way that would be  
41 analogous with the handling of such declarations in **FORTRAN 77**. Hence, mechanisms are provided  
42 through which all the information about a procedure’s interface may be made available in a scoping  
43 unit that references it. A procedure whose interface shall be deduced as in **FORTRAN 77** is described

1 as having an implicit interface. A procedure whose interface is fully known is described as having an  
 2 explicit interface.

3 A scoping unit is allowed to contain an interface body for a procedure that does not exist in the program,  
 4 provided the procedure described is never referenced or used in any other way. The purpose of this rule is  
 5 to allow implementations in which the use of a module providing interface bodies describing the interface  
 6 of every routine in a library would not automatically cause each of those library routines to be a part of  
 7 the program referencing the module. Instead, only those library procedures actually referenced would  
 8 be a part of the program. (In implementation terms, the mere presence of an interface body would not  
 9 generate an external reference in such an implementation.)

#### 10 **C.9.4 Abstract interfaces (12.4) and procedure pointer components (4.5)**

11 This is an example of a library module providing lists of callbacks that the user may register and invoke.

```

12 MODULE callback_list_module
13   !
14   ! Type for users to extend with their own data, if they so desire
15   !
16   TYPE callback_data
17   END TYPE
18   !
19   ! Abstract interface for the callback procedures
20   !
21   ABSTRACT INTERFACE
22     SUBROUTINE callback_procedure(data)
23       IMPORT callback_data
24       CLASS(callback_data),OPTIONAL :: data
25     END SUBROUTINE
26   END INTERFACE
27   !
28   ! The callback list type.
29   !
30   TYPE callback_list
31     PRIVATE
32     CLASS(callback_record),POINTER :: first => NULL()
33   END TYPE
34   !
35   ! Internal: each callback registration creates one of these
36   !
37   TYPE,PRIVATE :: callback_record
38     PROCEDURE(callback_procedure),POINTER,NOPASS :: proc
39     CLASS(callback_record),POINTER :: next
40     CLASS(callback_data),POINTER :: data => NULL();
41   END TYPE
42   PRIVATE invoke,forward_invoke
43 CONTAINS

```

```
1  !
2  ! Register a callback procedure with optional data
3  !
4  SUBROUTINE register_callback(list, entry, data)
5      TYPE(callback_list),INTENT(INOUT) :: list
6      PROCEDURE(callback_procedure) :: entry
7      CLASS(callback_data),OPTIONAL :: data
8      TYPE(callback_record),POINTER :: new,last
9      ALLOCATE(new)
10     new%proc => entry
11     IF (PRESENT(data)) ALLOCATE(new%data,SOURCE=data)
12     new%next => list%first
13     list%first => new
14 END SUBROUTINE
15 !
16 ! Internal: Invoke a single callback and destroy its record
17 !
18 SUBROUTINE invoke(callback)
19     TYPE(callback_record),POINTER :: callback
20     IF (ASSOCIATED(callback%data) THEN
21         CALL callback%proc(list%first%data)
22         DEALLOCATE(callback%data)
23     ELSE
24         CALL callback%proc
25     END IF
26     DEALLOCATE(callback)
27 END SUBROUTINE
28 !
29 ! Call the procedures in reverse order of registration
30 !
31 SUBROUTINE invoke_callback_reverse(list)
32     TYPE(callback_list),INTENT(INOUT) :: list
33     TYPE(callback_record),POINTER :: next,current
34     current => list%first
35     NULLIFY(list%first)
36     DO WHILE (ASSOCIATED(current))
37         next => current%next
38         CALL invoke(current)
39         current => next
40     END DO
41 END SUBROUTINE
42 !
43 ! Internal: Forward mode invocation
44 !
45 RECURSIVE SUBROUTINE forward_invoke(callback)
```

```

1      IF (ASSOCIATED(callback%next)) CALL forward_invoke(callback%next)
2      CALL invoke(callback)
3  END SUBROUTINE
4      !
5      ! Call the procedures in forward order of registration
6      !
7  SUBROUTINE invoke_callback_forward(list)
8      TYPE(callback_list), INTENT(INOUT) :: list
9      IF (ASSOCIATED(list%first)) CALL forward_invoke(list%first)
10  END SUBROUTINE
11 END

```

## 12 C.9.5 Argument association and evaluation (12.5.1.4)

13 There is a significant difference between the argument association allowed in this standard and that  
 14 supported by FORTRAN 77 and FORTRAN 66. In FORTRAN 77 and 66, actual arguments were limited  
 15 to consecutive storage units. With the exception of assumed length character dummy arguments, the  
 16 structure imposed on that sequence of storage units was always determined in the invoked procedure and  
 17 not taken from the actual argument. Thus it was possible to implement FORTRAN 66 and FORTRAN 77  
 18 argument association by supplying only the location of the first storage unit (except for character argu-  
 19 ments, where the length would also have to be supplied). However, this standard allows arguments that  
 20 do not reside in consecutive storage locations (for example, an array section), and dummy arguments that  
 21 assume additional structural information from the actual argument (for example, assumed-shape dummy  
 22 arguments). Thus, the mechanism to implement the argument association allowed in this standard needs  
 23 to be more general.

24 Because there are practical advantages to a processor that can support references to and from pro-  
 25 cedures defined by a FORTRAN 77 processor, requirements for explicit interfaces make it possible to  
 26 determine whether a simple (FORTRAN 66/FORTRAN 77) argument association implementation mecha-  
 27 nism is sufficient or whether the more general mechanism is necessary (12.4.1.1). Thus a processor can  
 28 be implemented whose procedures expect the simple mechanism to be used whenever the procedure's  
 29 interface is one that uses only FORTRAN 77 features and that expects the more general mechanism  
 30 otherwise (for example, if there are assumed-shape or optional arguments). At the point of reference,  
 31 the appropriate mechanism can be determined from the interface if it is explicit and can be assumed to  
 32 be the simple mechanism if it is not. Note that if the simple mechanism is determined to be what the  
 33 procedure expects, it may be necessary for the processor to allocate consecutive temporary storage for  
 34 the actual argument, copy the actual argument to the temporary storage, reference the procedure using  
 35 the temporary storage in place of the actual argument, copy the contents of temporary storage back to  
 36 the actual argument, and deallocate the temporary storage.

37 While this is the particular implementation method these rules were designed to support, it is not  
 38 the only one possible. For example, on some processors, it may be possible to implement the general  
 39 argument association in such a way that the information involved in FORTRAN 77 argument association  
 40 may be found in the same places and the "extra" information is placed so it does not disturb a procedure  
 41 expecting only FORTRAN 77 argument association. With such an implementation, argument association  
 42 could be translated without regard to whether the interface is explicit or implicit.

43 The provisions for expression evaluation give the processor considerable flexibility for obtaining expres-  
 44 sion values in the most efficient way possible. This includes not evaluating or only partially evaluating  
 45 an operand, for example, if the value of the expression can be determined otherwise (7.1.8.1). This  
 46 flexibility applies to function argument evaluation, including the order of argument evaluation, delay-  
 47 ing argument evaluation, and omitting argument evaluation. A processor may delay the evaluation of

1 an argument in a procedure reference until the execution of the procedure refers to the value of that  
 2 argument, provided delaying the evaluation of the argument does not otherwise affect the results of  
 3 the program. The processor may, with similar restrictions, entirely omit the evaluation of an argument  
 4 not referenced in the execution of the procedure. This gives processors latitude for optimization (for  
 5 example, for parallel processing).

#### 6 **C.9.6 Pointers and targets as arguments (12.5.1.4)**

7 If a dummy argument is declared to be a pointer, it may be matched only by an actual argument that  
 8 also is a pointer, and the characteristics of both arguments shall agree. A model for such an association is  
 9 that descriptor values of the actual pointer are copied to the dummy pointer. If the actual pointer has an  
 10 associated target, this target becomes accessible via the dummy pointer. If the dummy pointer becomes  
 11 associated with a different target during execution of the procedure, this target will be accessible via the  
 12 actual pointer after the procedure completes execution. If the dummy pointer becomes associated with  
 13 a local target that ceases to exist when the procedure completes, the actual pointer will be left dangling  
 14 in an undefined state. Such dangling pointers shall not be used.

15 When execution of a procedure completes, any pointer that remains defined and that is associated with  
 16 a dummy argument that has the TARGET attribute and is either a scalar or an assumed-shape array,  
 17 remains associated with the corresponding actual argument if the actual argument has the TARGET  
 18 attribute and is not an array section with a vector subscript.

```

19 REAL, POINTER      :: PBEST
20 REAL, TARGET       :: B (10000)
21 CALL BEST (PBEST, B)           ! Upon return PBEST is associated
22     ...                       ! with the 'best' element of B
23 CONTAINS
24   SUBROUTINE BEST (P, A)
25     REAL, POINTER, INTENT (OUT) :: P
26     REAL, TARGET, INTENT (IN)   :: A (:)
27     ...                         ! Find the 'best' element A(I)
28     P => A (I)
29   RETURN
30 END SUBROUTINE BEST
31 END

```

32 When procedure BEST completes, the pointer PBEST is associated with an element of B.

33 An actual argument without the TARGET attribute can become associated with a dummy argument  
 34 with the TARGET attribute. This permits pointers to become associated with the dummy argument  
 35 during execution of the procedure that contains the dummy argument. For example:

```

36 INTEGER LARGE(100,100)
37 CALL SUB (LARGE)
38     ...
39 CALL SUB ( )
40 CONTAINS
41   SUBROUTINE SUB(ARG)
42     INTEGER, TARGET, OPTIONAL :: ARG(100,100)

```

```

1    INTEGER, POINTER, DIMENSION(:,:) :: PARG
2    IF (PRESENT(ARG)) THEN
3        PARG => ARG
4    ELSE
5        ALLOCATE (PARG(100,100))
6        PARG = 0
7    ENDIF
8        ... ! Code with lots of references to PARG
9    IF (.NOT. PRESENT(ARG)) DEALLOCATE(PARG)
10   END SUBROUTINE SUB
11   END

```

12 Within subroutine SUB the pointer PARG is either associated with the dummy argument ARG or it is  
 13 associated with an allocated target. The bulk of the code can reference PARG without further calls to  
 14 the PRESENT intrinsic.

### 15 C.9.7 Polymorphic Argument Association (12.5.1.6)

16 The following example illustrates polymorphic argument association rules using the derived types defined  
 17 in Note 4.58.

```

18   TYPE(POINT) :: T2
19   TYPE(COLOR_POINT) :: T3
20   CLASS(POINT) :: P2
21   CLASS(COLOR_POINT) :: P3
22   ! Dummy argument is polymorphic and actual argument is of fixed type
23   SUBROUTINE SUB2 ( X2 ); CLASS(POINT) :: X2; ...
24   SUBROUTINE SUB3 ( X3 ); CLASS(COLOR_POINT) :: X3; ...
25
26   CALL SUB2 ( T2 ) ! Valid -- The declared type of T2 is the same as the
27                   !           declared type of X2.
28   CALL SUB2 ( T3 ) ! Valid -- The declared type of T3 is extended from
29                   !           the declared type of X2.
30   CALL SUB3 ( T2 ) ! Invalid -- The declared type of T2 is neither the
31                   !           same as nor extended from the declared type
32                   !           type of X3.
33   CALL SUB3 ( T3 ) ! Valid -- The declared type of T3 is the same as the
34                   !           declared type of X3.
35   ! Actual argument is polymorphic and dummy argument is of fixed type
36   SUBROUTINE TUB2 ( D2 ); TYPE(POINT) :: D2; ...
37   SUBROUTINE TUB3 ( D3 ); TYPE(COLOR_POINT) :: D3; ...
38
39   CALL TUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
40                   !           declared type of D2.
41   CALL TUB2 ( P3 ) ! Invalid -- The declared type of P3 differs from the
42                   !           declared type of D2.

```

```

1  CALL TUB2 ( P3%POINT ) ! Valid alternative to the above
2  CALL TUB3 ( P2 ) ! Invalid -- The declared type of P2 differs from the
3      ! declared type of D3.
4  SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
5  CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the same
6      CALL TUB3 ( P2 ) ! as the declared type of D3, or a type
7      ! extended therefrom.
8  CLASS DEFAULT
9      ! Cannot work if not.
10 END SELECT
11 CALL TUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
12     ! declared type of D3.
13 ! Both the actual and dummy arguments are of polymorphic type.
14 CALL SUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
15     ! declared type of X2.
16 CALL SUB2 ( P3 ) ! Valid -- The declared type of P3 is extended from
17     ! the declared type of X2.
18 CALL SUB3 ( P2 ) ! Invalid -- The declared type of P2 is neither the
19     ! same as nor extended from the declared
20     ! type of X3.
21 SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
22 CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the
23     CALL SUB3 ( P2 ) ! same as the declared type of X3, or a
24     ! type extended therefrom.
25 CLASS DEFAULT
26     ! Cannot work if not.
27 END SELECT
28 CALL SUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
29     ! declared type of X3.

```

## 30 C.10 Clause 13 notes

### 31 C.10.1 Module for THIS\_IMAGE and IMAGE\_INDEX

32 The intrinsics THIS\_IMAGE (CO\_ARRAY) and IMAGE\_INDEX (CO\_ARRAY, SUB) cannot be written  
 33 in Fortran since CO\_ARRAY may be of any type and THIS\_IMAGE (CO\_ARRAY) needs to know the  
 34 index of the image on which the code is running.

35 As an example, here are simple versions that require the co-bounds to be specified as integer arrays and  
 36 require the image index for THIS\_IMAGE (CO\_ARRAY).

```

37 MODULE index
38 CONTAINS
39     INTEGER FUNCTION image_index(lbound, ubound, sub)
40         INTEGER, INTENT(IN) :: lbound(:), ubound(:), sub(:)
41         INTEGER :: i, n

```

```

1      n = SIZE(sub)
2      image_index = sub(n) - lbound(n)
3      DO i = n-1, 1, -1
4          image_index = image_index*(ubound(i)-lbound(i)+1) + sub(i) - lbound(i)
5      END DO
6      image_index = image_index + 1
7  END FUNCTION image_index
8
9  INTEGER FUNCTION this_image(lbound, ubound, me) RESULT(sub)
10     INTEGER, INTENT(IN) :: lbound(:), ubound(:), me
11     INTEGER              :: sub(SIZE(lbound))
12     INTEGER              :: extent, i, m, ml, n
13     n = SIZE(sub)
14     m = me - 1
15     DO i = 1, n-1
16         extent = ubound(i) - lbound(i) + 1
17         ml = m
18         m = m/extent
19         sub(i) = ml - m*extent + lbound(i)
20     END DO
21     sub(n) = m + lbound(n)
22 END FUNCTION this_IMAGE
23 END MODULE index

```

## 24 C.10.2 Collective co-array subroutine variations

25 The subroutines of 13.5.15 return an array of the same shape as the given co-array after having applied  
26 an operation on the images involved. A note there shows how to program the operation being applied to  
27 all the elements of the co-array to produce a scalar result. Versions with other arguments can similarly  
28 be programmed, for example:

```

29 MODULE global_sum_module
30     INTRINSIC, PRIVATE :: CO_SUM, SIZE, SUM
31 CONTAINS
32     REAL FUNCTION global_sum_mask(array, mask)
33         REAL, INTENT(IN) :: array(:, :)[*], mask(:, :)
34         REAL, SAVE       :: temp[*]
35         temp = SUM(array, MASK=mask) ! Sum of the local part of the co-array.
36         CALL CO_SUM(temp, global_sum_mask)
37     END FUNCTION global_sum_mask
38
39     FUNCTION global_sum(array, dim)
40         REAL, INTENT(IN) :: array(:, :)[*]
41         INTEGER, INTENT(IN) :: dim
42         REAL, ALLOCATABLE :: global_sum(:)
43         REAL, ALLOCATABLE :: temp(:)[:]

```



```

1      ALLOCATE (global_sum(SIZE(array, 3-dim)))
2      ALLOCATE (      temp(SIZE(array, 3-dim))[*])
3      temp = SUM(array, dim) ! Sum of the local part of the co-array.
4      CALL CO_SUM(temp, global_sum)
5  END FUNCTION global_sum
6 END MODULE global_sum_module

```

## 7 C.11 Clause 15 notes

### 8 C.11.1 Runtime environments

9 This standard allows programs to contain procedures defined by means other than Fortran. That raises  
10 the issues of initialization of and interaction between the runtime environments involved.

11 Implementations are free to solve these issues as they see fit, provided that

- 12 (1) heap allocation/deallocation (e.g., (DE)ALLOCATE in a Fortran subprogram and mal-  
13 loc/free in a C function) can be performed without interference,
- 14 (2) input/output to and from external files can be performed without interference, as long as  
15 procedures defined by different means do not do input/output with the same external file,
- 16 (3) input/output preconnections exist as required by the respective standards, and
- 17 (4) initialized data is initialized according to the respective standards.

### 18 C.11.2 Examples of Interoperation between Fortran and C Functions

19 The following examples illustrate the interoperation of Fortran and C functions. Two examples are  
20 shown: one of Fortran calling C, and one of C calling Fortran. In each of the examples, the correspon-  
21 dences of Fortran actual arguments, Fortran dummy arguments, and C formal parameters are described.

#### 22 C.11.2.1 Example of Fortran calling C

23 C Function Prototype:

```

24      int C_Library_Function(void* sendbuf, int sendcount,
25                          int *recvcounts);

```

26 Fortran Modules:

```

27      MODULE FTN_C_1
28          USE, INTRINSIC :: ISO_C_BINDING
29      END MODULE FTN_C_1
30
31      MODULE FTN_C_2
32          INTERFACE
33
34              INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
              (SENDBUF, SENDCOUNT, RECVCOUNTS) &

```

```

1      BIND(C, NAME='C_Library_Function')
2      USE FTN_C_1
3      IMPLICIT NONE
4      TYPE (C_PTR), VALUE :: SENDBUF
5      INTEGER (C_INT), VALUE :: SENDCOUNT
6      TYPE (C_PTR), VALUE :: RECVCOUNTS
7      END FUNCTION C_LIBRARY_FUNCTION
8      END INTERFACE
9      END MODULE FTN_C_2

```

10 The module FTN\_C\_2 contains the declaration of the Fortran dummy arguments, which correspond to  
 11 the C formal parameters. The intrinsic module ISO\_C\_BINDING is referenced in the module FTN\_C\_1.  
 12 The NAME specifier is used in the BIND attribute in order to handle the case-sensitive name change  
 13 between Fortran and C from 'C\_LIBRARY\_FUNCTION' to 'C\_Library\_Function'. See also Note 12.43.

14 The first C formal parameter is the pointer to void `sendbuf`, which corresponds to the Fortran dummy  
 15 argument `SENDBUF`, which has the type `C_PTR` and the `VALUE` attribute.

16 The second C formal parameter is the int `sendcount`, which corresponds to the Fortran dummy argument  
 17 `SENDCOUNT`, which has the type `INTEGER(C_INT)` and the `VALUE` attribute.

18 The third C formal parameter is the pointer to int `recvcunts`, which corresponds to the Fortran dummy  
 19 argument `RECVCOUNTS`, which has the type `C_PTR` and the `VALUE` attribute.

20 Fortran Calling Sequence:

```

21      USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
22      USE FTN_C_2
23      ...
24      REAL (C_FLOAT), TARGET  :: SEND(100)
25      INTEGER (C_INT)         :: SENDCOUNT
26      INTEGER (C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(100)
27      ...
28      ALLOCATE( RECVCOUNTS(100) )
29      ...
30      CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
31      C_LOC(RECVCOUNTS))
32      ...

```

33 The preceding code contains the declaration of the Fortran actual arguments associated with the above-  
 34 listed Fortran dummy arguments.

35 The first Fortran actual argument is the address of the first element of the array `SEND`, which has the  
 36 type `REAL(C_FLOAT)` and the `TARGET` attribute. This address is returned by the intrinsic function  
 37 `C_LOC`. This actual argument is associated with the Fortran dummy argument `SENDBUF`, which has  
 38 the type `C_PTR` and the `VALUE` attribute.

39 The second Fortran actual argument is `SENDCOUNT` of type `INTEGER(C_INT)`, which is associated  
 40 with the Fortran dummy argument `SENDCOUNT`, which has the type `INTEGER(C_INT)` and the  
 41 `VALUE` attribute.

1 The third Fortran actual argument is the address of the first element of the allocatable array RECV-  
 2 COUNTS, with has the type REAL(C\_FLOAT) and the TARGET attribute. This address is returned  
 3 by the intrinsic function C\_LOC. This actual argument is associated with the Fortran dummy argument  
 4 RECVCOUNTS, which has the type C\_PTR and the VALUE attribute.

### 5 C.11.2.2 Example of C calling Fortran

6 Fortran Code:

```

7 SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS) BIND(C)
8   USE, INTRINSIC :: ISO_C_BINDING
9   IMPLICIT NONE
10  INTEGER (C_LONG), VALUE           :: ALPHA
11  REAL (C_DOUBLE), INTENT(INOUT)    :: BETA
12  INTEGER (C_LONG), INTENT(OUT)     :: GAMMA
13  REAL (C_DOUBLE), DIMENSION(*), INTENT(IN) :: DELTA
14  TYPE, BIND(C) :: PASS
15     INTEGER (C_INT) :: LENC, LENF
16     TYPE (C_PTR)    :: C, F
17  END TYPE PASS
18  TYPE (PASS), INTENT(INOUT) :: ARRAYS
19  REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
20  REAL (C_FLOAT), POINTER :: C_ARRAY(:)
21  ...
22  ! Associate C_ARRAY with an array allocated in C
23  CALL C_F_POINTER (ARRAYS%C, C_ARRAY, (/ARRAYS%LENC/))
24  ...
25  ! Allocate an array and make it available in C
26  ARRAYS%LENF = 100
27  ALLOCATE (ETA(ARRAYS%LENF))
28  ARRAYS%F = C_LOC(ETA)
29  ...
30 END SUBROUTINE SIMULATION

```

31 C Struct Declaration

```

32     struct pass {int lenc, lenf; float *c, *f;};

```

33 C Function Prototype:

```

34     void simulation(long alpha, double *beta, long *gamma,
35         double delta[], struct pass *arrays);

```

36 C Calling Sequence:

```

37     simulation(alpha, &beta, &gamma, delta, &arrays);

```

- 1 The above-listed Fortran code specifies a subroutine SIMULATION. This subroutine corresponds to the  
 2 C void function `simulation`.
- 3 The Fortran subroutine references the intrinsic module ISO\_C\_BINDING.
- 4 The first Fortran dummy argument of the subroutine is ALPHA, which has the type INTEGER(C\_-  
 5 LONG) and the attribute VALUE. This dummy argument corresponds to the C formal parameter  
 6 `alpha`, which is a long. The actual C parameter is also a long.
- 7 The second Fortran dummy argument of the subroutine is BETA, which has the type REAL(C\_-  
 8 DOUBLE) and the INTENT(INOUT) attribute. This dummy argument corresponds to the C formal  
 9 parameter `beta`, which is a pointer to double. An address is passed as the actual parameter in the C  
 10 calling sequence.
- 11 The third Fortran dummy argument of the subroutine is GAMMA, which has the type INTEGER(C\_-  
 12 LONG) and the INTENT(OUT) attribute. This dummy argument corresponds to the C formal param-  
 13 eter `gamma`, which is a pointer to long. An address is passed as the actual parameter in the C calling  
 14 sequence.
- 15 The fourth Fortran dummy argument is the assumed-size array DELTA, which has the type REAL  
 16 (C\_DOUBLE) and the attribute INTENT(IN). This dummy argument corresponds to the C formal  
 17 parameter `delta`, which is a double array. The actual C parameter is also a double array.
- 18 The fifth Fortran dummy argument is ARRAYS, which is a structure for accessing an array allocated  
 19 in C and an array allocated in Fortran. The lengths of these arrays are held in the components LENC  
 20 and LENF; their C addresses are held in components C and F.

### 21 C.11.2.3 Example of calling C functions with noninteroperable data

- 22 Many Fortran processors support 16-byte real numbers, which might not be supported by the C processor.  
 23 Assume a Fortran programmer wants to use a C procedure from a message passing library for an array  
 24 of these reals. The C prototype of this procedure is

```
25 void ProcessBuffer(void *buffer, int n_bytes);
```

- 26 with the corresponding Fortran interface

```
27 USE, INTRINSIC :: ISO_C_BINDING
28
29 INTERFACE
30     SUBROUTINE PROCESS_BUFFER(BUFFER,N_BYTES) BIND(C,NAME="ProcessBuffer")
31         IMPORT :: C_PTR, C_INT
32         TYPE(C_PTR), VALUE :: BUFFER ! The 'C address' of the array buffer
33         INTEGER(C_INT), VALUE :: N_BYTES ! Number of bytes in buffer
34     END SUBROUTINE PROCESS_BUFFER
35 END INTERFACE
```

- 36 This may be done using C\_LOC if the particular Fortran processor specifies that C\_LOC returns an  
 37 appropriate address:

```
38 REAL(R_QUAD), DIMENSION(:), ALLOCATABLE, TARGET :: QUAD_ARRAY
```

```

1  ...
2  CALL PROCESS_BUFFER(C_LOC(QUAD_ARRAY), INT(16*SIZE(QUAD_ARRAY),C_INT))
3      ! One quad real takes 16 bytes on this processor

```

#### 4 C.11.2.4 Example of opaque communication between C and Fortran

5 The following example demonstrates how a Fortran processor can make a modern OO random number  
6 generator written in Fortran available to a C program:

```

7  USE, INTRINSIC :: ISO_C_BINDING
8      ! Assume this code is inside a module
9
10 TYPE RANDOM_STREAM
11     ! A (uniform) random number generator (URNG)
12 CONTAINS
13     PROCEDURE(RANDOM_UNIFORM), DEFERRED, PASS(STREAM) :: NEXT
14     ! Generates the next number from the stream
15 END TYPE RANDOM_STREAM
16
17 ABSTRACT INTERFACE
18     ! Abstract interface of Fortran URNG
19     SUBROUTINE RANDOM_UNIFORM(STREAM, NUMBER)
20         IMPORT :: RANDOM_STREAM, C_DOUBLE
21         CLASS(RANDOM_STREAM), INTENT(INOUT) :: STREAM
22         REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
23     END SUBROUTINE RANDOM_UNIFORM
24 END INTERFACE

```

25 A polymorphic object of base type RANDOM\_STREAM is not interoperable with C. However, we can  
26 make such a random number generator available to C by packaging it inside another nonpolymorphic,  
27 nonparameterized derived type:

```

28 TYPE :: URNG_STATE ! No BIND(C), as this type is not interoperable
29     CLASS(RANDOM_STREAM), ALLOCATABLE :: STREAM
30 END TYPE URNG_STATE

```

31 The following two procedures will enable a C program to use our Fortran uniform random number  
32 generator:

```

33 ! Initialize a uniform random number generator:
34 SUBROUTINE INITIALIZE_URNG(STATE_HANDLE, METHOD) &
35     BIND(C, NAME="InitializeURNG")
36     TYPE(C_PTR), INTENT(OUT) :: STATE_HANDLE
37     ! An opaque handle for the URNG
38     CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: METHOD
39     ! The algorithm to be used

```

```

1
2     TYPE(URNG_STATE), POINTER :: STATE
3         ! An actual URNG object
4
5     ALLOCATE(STATE)
6         ! There needs to be a corresponding finalization
7         ! procedure to avoid memory leaks, not shown in this example
8         ! Allocate STATE%STREAM with a dynamic type depending on METHOD
9         ...
10    STATE_HANDLE=C_LOC(STATE)
11        ! Obtain an opaque handle to return to C
12 END SUBROUTINE INITIALIZE_URNG
13
14 ! Generate a random number:
15 SUBROUTINE GENERATE_UNIFORM(STATE_HANDLE, NUMBER) &
16     BIND(C, NAME="GenerateUniform")
17     TYPE(C_PTR), INTENT(IN), VALUE :: STATE_HANDLE
18         ! An opaque handle: Obtained via a call to INITIALIZE_URNG
19     REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
20
21     TYPE(URNG_STATE), POINTER :: STATE
22         ! A pointer to the actual URNG
23
24     CALL C_F_POINTER(CPTR=STATE_HANDLE, FPTR=STATE)
25         ! Convert the opaque handle into a usable pointer
26     CALL STATE%STREAM%NEXT(NUMBER)
27         ! Use the type-bound procedure NEXT to generate NUMBER
28 END SUBROUTINE GENERATE_UNIFORM

```

## 29 C.12 Clause 16 notes

### 30 C.12.1 Examples of host association (16.5.1.4)

31 The first two examples are examples of valid host association. The third example is an example of invalid  
 32 host association.

33 Example 1:

```

34 PROGRAM A
35     INTEGER I, J
36     ...
37 CONTAINS
38     SUBROUTINE B
39         INTEGER I ! Declaration of I hides
40                 ! program A's declaration of I
41         ...

```

```

1      I = J      ! Use of variable J from program A
2                  ! through host association
3      END SUBROUTINE B
4  END PROGRAM A

```

5 Example 2:

```

6  PROGRAM A
7      TYPE T
8          ...
9      END TYPE T
10     ...
11  CONTAINS
12      SUBROUTINE B
13          IMPLICIT TYPE (T) (C)  ! Refers to type T declared below
14                                  ! in subroutine B, not type T
15                                  ! declared above in program A
16          ...
17          TYPE T
18              ...
19          END TYPE T
20          ...
21      END SUBROUTINE B
22  END PROGRAM A

```

23 Example 3:

```

24  PROGRAM Q
25      REAL (KIND = 1) :: C
26      ...
27  CONTAINS
28      SUBROUTINE R
29          REAL (KIND = KIND (C)) :: D  ! Invalid declaration
30                                          ! See below
31          REAL (KIND = 2) :: C
32          ...
33      END SUBROUTINE R
34  END PROGRAM Q

```

35 In the declaration of D in subroutine R, the use of C would refer to the declaration of C in subroutine  
36 R, not program Q. However, it is invalid because the declaration of C is required to occur before it is  
37 used in the declaration of D (7.1.7).

### 38 **C.12.2 Rules ensuring unambiguous generics (16.3.4)**

39 The rules in 16.3.4 are intended to ensure

- 1     • that it is possible to reference each specific procedure in the generic collection,
- 2     • that for any valid reference to the generic procedure, the determination of the specific procedure
- 3       referenced is unambiguous, and
- 4     • that the determination of the specific procedure referenced can be made before execution of the
- 5       program begins (during compilation).

6 Specific procedures are distinguished by fixed properties of their arguments, specifically type, kind type  
 7 parameters, and rank. A valid reference to one procedure in a generic collection will differ from another  
 8 because it has an argument that the other cannot accept, because it is missing an argument that the  
 9 other requires, or because one of these fixed properties is different.

10 Although the declared type of a data entity is a fixed property, polymorphic variables allow for a  
 11 limited degree of type mismatch between dummy arguments and actual arguments, so the requirement  
 12 for distinguishing two dummy arguments is type incompatibility, not merely different types. (This is  
 13 illustrated in the BAD6 example later in this note.)

14 That same limited type mismatch means that two dummy arguments that are not type incompatible  
 15 can be distinguished on the basis of the values of the kind type parameters they have in common; if one  
 16 of them has a kind type parameter that the other does not, that is irrelevant in distinguishing them.

17 Rank is a fixed property, but some forms of array dummy arguments allow rank mismatches when a  
 18 procedure is referenced by its specific name. In order to allow rank to always be usable in distinguishing  
 19 generics, such rank mismatches are disallowed for those arguments when the procedure is referenced as  
 20 part of a generic. Additionally, the fact that elemental procedures can accept array arguments is not  
 21 taken into account when applying these rules, so apparent ambiguity between elemental and nonelemental  
 22 procedures is possible; in such cases, the reference is interpreted as being to the nonelemental procedure.

23 For procedures referenced as operators or defined-assignment, syntactically distinguished arguments are  
 24 mapped to specific positions in the argument list, so the rule for distinguishing such procedures is that  
 25 it be possible to distinguish the arguments at one of the argument positions.

26 For user-defined derived-type input/output procedures, only the `dtv` argument corresponds to something  
 27 explicitly written in the program, so it is the `dtv` that is required to be distinguished. Because `dtv`  
 28 arguments are required to be scalar, they cannot differ in rank. Thus this rule effectively involves only  
 29 type and kind type parameters.

30 For generic procedures identified by names, the rules are more complicated because optional arguments  
 31 may be omitted and because arguments may be specified either positionally or by name.

32 In the special case of type-bound procedures with passed-object dummy arguments, the passed-object  
 33 argument is syntactically distinguished in the reference, so rule (2) can be applied. The type of passed-  
 34 object arguments is constrained in ways that prevent passed-object arguments in the same scoping unit  
 35 from being type incompatible. Thus this rule effectively involves only kind type parameters and rank.

36 The primary means of distinguishing named generics is rule (3). The most common application of that  
 37 rule is a single argument satisfying both (3a) and (3b):

```

38     INTERFACE GOOD1
39         FUNCTION F1A(X)
40             REAL :: F1A,X
41         END FUNCTION F1A
42         FUNCTION F1B(X)
43             INTEGER :: F1B,X

```



```

1          END FUNCTION F1B
2          END INTERFACE GOOD1

```

3 Whether one writes `GOOD1(1.0)` or `GOOD1(X=1.0)`, the reference is to `F1A` because `F1B` would require an  
4 integer argument whereas these references provide the real constant 1.0.

5 This example and those that follow are expressed using interface bodies, with type as the distinguishing  
6 property. This was done to make it easier to write and describe the examples. The principles being  
7 illustrated are equally applicable when the procedures get their explicit interfaces in some other way or  
8 when kind type parameters or rank are the distinguishing property.

9 Another common variant is the argument that satisfies (3a) and (3b) by being required in one specific  
10 and completely missing in the other:

```

11         INTERFACE GOOD2
12             FUNCTION F2A(X)
13                 REAL :: F2A,X
14             END FUNCTION F2A
15             FUNCTION F2B(X,Y)
16                 COMPLEX :: F2B
17                 REAL :: X,Y
18             END FUNCTION F2B
19         END INTERFACE GOOD2

```

20 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  
21 `F2B`, because `F2A` has no argument in the second position or with the name `Y`. This approach is used as  
22 an alternative to optional arguments when one wants a function to have different result type, kind type  
23 parameters, or rank, depending on whether the argument is present. In many of the intrinsic functions,  
24 the `DIM` argument works this way.

25 It is possible to construct cases where different arguments are used to distinguish positionally and by  
26 name:

```

27         INTERFACE GOOD3
28             SUBROUTINE S3A(W,X,Y,Z)
29                 REAL :: W,Y
30                 INTEGER :: X,Z
31             END SUBROUTINE S3A
32             SUBROUTINE S3B(X,W,Z,Y)
33                 REAL :: W,Z
34                 INTEGER :: X,Y
35             END SUBROUTINE S3B
36         END INTERFACE GOOD3

```

37 If one writes `GOOD3(1.0,2,3.0,4)` to reference `S3A`, then the third and fourth arguments are consistent  
38 with a reference to `S3B`, but the first and second are not. If one switches to writing the first two  
39 arguments as keyword arguments in order for them to be consistent with a reference to `S3B`, the latter  
40 two arguments must also be written as keyword arguments, `GOOD3(X=2,W= 1.0,Z=4,Y=3.0)`, and the  
41 named arguments `Y` and `Z` are distinguished.

1 The ordering requirement in rule (3) is critical:

```

2      INTERFACE BAD4  ! this interface is invalid !
3          SUBROUTINE S4A(W,X,Y,Z)
4              REAL :: W,Y
5              INTEGER :: X,Z
6          END SUBROUTINE S4A
7          SUBROUTINE S4B(X,W,Z,Y)
8              REAL :: X,Y
9              INTEGER :: W,Z
10         END SUBROUTINE S4B
11     END INTERFACE BAD4

```

12 In this example, the positionally distinguished arguments are Y and Z, and it is W and X that are  
13 distinguished by name. In this order it is possible to write `BAD4(1.0,2,Y=3.0,Z=4)`, which is a valid  
14 reference for both `S4A` and `S4B`.

15 Rule (1) can be used to distinguish some cases that are not covered by rule (3):

```

16     INTERFACE GOOD5
17         SUBROUTINE S5A(X)
18             REAL :: X
19         END SUBROUTINE S5A
20         SUBROUTINE S5B(Y,X)
21             REAL :: Y,X
22         END SUBROUTINE S5B
23     END INTERFACE GOOD5

```

24 In attempting to apply rule (3), position 2 and name Y are distinguished, but they are in the wrong  
25 order, just like the `BAD4` example. However, when we try to construct a similarly ambiguous reference,  
26 we get `GOOD5(1.0,X=2.0)`, which can't be a reference to `S5A` because it would be attempting to associate  
27 two different actual arguments with the dummy argument X. Rule (3) catches this case by recognizing  
28 that `S5B` requires two real arguments, and `S5A` cannot possibly accept more than one.

29 The application of rule (1) becomes more complicated when extensible types are involved. If `FRUIT` is  
30 an extensible type, `PEAR` and `APPLE` are extensions of `FRUIT`, and `BOSC` is an extension of `PEAR`, then

```

31     INTERFACE BAD6  ! this interface is invalid !
32         SUBROUTINE S6A(X,Y)
33             CLASS(PEAR) :: X,Y
34         END SUBROUTINE S6A
35         SUBROUTINE S6B(X,Y)
36             CLASS(FRUIT) :: X
37             CLASS(BOSC) :: Y
38         END SUBROUTINE S6B
39     END INTERFACE BAD6

```

1 might, at first glance, seem distinguishable this way, but because of the limited type mismatching allowed,  
 2 BAD6(A\_PEAR,A\_BOSC) is a valid reference to both S6A and S6B.

3 It is important to try rule (1) for each type present:

```

4      INTERFACE GOOD7
5          SUBROUTINE S7A(X,Y,Z)
6              CLASS(PEAR) :: X,Y,Z
7          END SUBROUTINE S7A
8          SUBROUTINE S7B(X,Z,W)
9              CLASS(FRUIT) :: X
10             CLASS(BOSC) :: Z
11             CLASS(APPLE),OPTIONAL :: W
12          END SUBROUTINE S7B
13      END INTERFACE GOOD7

```

14 Looking at the most general type, S7A has a minimum and maximum of 3 FRUIT arguments, while S7B  
 15 has a minimum of 2 and a maximum of three. Looking at the most specific, S7A has a minimum of 0  
 16 and a maximum of 3 BOSC arguments, while S7B has a minimum of 1 and a maximum of 2. However,  
 17 when we look at the intermediate, S7A has a minimum and maximum of 3 PEAR arguments, while S7B  
 18 has a minimum of 1 and a maximum of 2. Because S7A's minimum exceeds S7B's maximum, they can  
 19 be distinguished.

20 In identifying the minimum number of arguments with a particular set of properties, we exclude optional  
 21 arguments and test TKR compatibility, so the corresponding actual arguments are required to have  
 22 those properties. In identifying the maximum number of arguments with those properties, we include  
 23 the optional arguments and test not distinguishable, so we include actual arguments which could have  
 24 those properties but are not required to have them.

25 These rules are sufficient to ensure that references to procedures that meet them are unambiguous, but  
 26 there remain examples that fail to meet these rules but which can be shown to be unambiguous:

```

27      INTERFACE BAD8 ! this interface is invalid !
28      ! despite the fact that it is unambiguous !
29          SUBROUTINE S8A(X,Y,Z)
30              REAL,OPTIONAL :: X
31              INTEGER :: Y
32              REAL :: Z
33          END SUBROUTINE S8A
34          SUBROUTINE S8B(X,Z,Y)
35              INTEGER,OPTIONAL :: X
36              INTEGER :: Z
37              REAL :: Y
38          END SUBROUTINE S8B
39      END INTERFACE BAD8

```

40 This interface fails rule (3) because there are no required arguments that can be distinguished from the  
 41 positionally corresponding argument, but in order for the mismatch of the optional arguments not to  
 42 be relevant, the later arguments must be specified as keyword arguments, so distinguishing by name

- 1 does the trick. This interface is nevertheless invalid so a standard- conforming Fortran processor is not  
 2 required to do such reasoning. The rules to cover all cases are too complicated to be useful.
- 3 In addition to not recognizing distinguishable patterns like the one in BAD8, the rules do not distinguish  
 4 on the basis of any properties other than type, kind type parameters, and rank:

```

5      INTERFACE BAD9 ! this interface is invalid !
6      ! despite the fact that it is unambiguous !
7      SUBROUTINE S9A(X)
8          REAL :: X
9      END SUBROUTINE S9A
10     SUBROUTINE S9B(X)
11         INTERFACE
12             FUNCTION X(A)
13                 REAL :: X,A
14             END FUNCTION X
15         END INTERFACE
16     END SUBROUTINE S9B
17     SUBROUTINE S9C(X)
18         INTERFACE
19             FUNCTION X(A)
20                 REAL :: X
21                 INTEGER :: A
22             END FUNCTION X
23         END INTERFACE
24     END SUBROUTINE S9C
25 END INTERFACE BAD9

```

- 26 The real data objects that would be valid arguments for S9A are entirely disjoint from procedures that  
 27 are valid arguments to S9B and S9C, and the procedures that valid arguments for S9B are disjoint from  
 28 the procedures that are valid arguments to S9C because the former are required to accept real arguments  
 29 and the latter integer arguments. Again, this interface is invalid, so a standard-conforming Fortran  
 30 processor need not examine such properties when deciding whether a generic collection is valid. Again,  
 31 the rules to cover all cases are too complicated to be useful.

## 32 C.13 Array feature notes

### 33 C.13.1 Summary of features

- 34 This subclause is a summary of the principal array features.

#### 35 C.13.1.1 Whole array expressions and assignments (7.4.1.2, 7.4.1.3)

- 36 An important feature is that whole array expressions and assignments are permitted. For example, the  
 37 statement

38 `A = B + C * SIN (D)`

- 39 where A, B, C, and D are arrays of the same shape, is permitted. It is interpreted element-by-element;

1 that is, the sine function is taken on each element of D, each result is multiplied by the corresponding  
 2 element of C, added to the corresponding element of B, and assigned to the corresponding element of  
 3 A. Functions, including user-written functions, may be arrays and may be generic with scalar versions.  
 4 All arrays in an expression or across an assignment shall conform; that is, have exactly the same shape  
 5 (number of dimensions and extents in each dimension), but scalars may be included freely and these are  
 6 interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment  
 7 takes place.

#### 8 **C.13.1.2 Array sections (2.4.5, 6.2.2.3)**

9 Whenever whole arrays may be used, it is also possible to use subarrays called “sections”. For example:

10 `A (:, 1:N, 2, 3:1:-1)`

11 consists of a subarray containing the whole of the first dimension, positions 1 to N of the second dimen-  
 12 sion, position 2 of the third dimension and positions 1 to 3 in reverse order of the fourth dimension.  
 13 This is an artificial example chosen to illustrate the different forms. Of course, a common use may be  
 14 to select a row or column of an array, for example:

15 `A (:, J)`

#### 16 **C.13.1.3 WHERE statement (7.4.3)**

17 The WHERE statement applies a conforming logical array as a mask on the individual operations in the  
 18 expression and in the assignment. For example:

19 `WHERE (A > 0) B = LOG (A)`

20 takes the logarithm only for positive components of A and makes assignments only in these positions.

21 The WHERE statement also has a block form (WHERE construct).

#### 22 **C.13.1.4 Automatic arrays and allocatable variables (5.2, 5.3.7.4)**

23 Two features useful for writing modular software are automatic arrays, created on entry to a subprogram  
 24 and destroyed on return, and allocatable variables, including arrays whose rank is fixed but whose actual  
 25 size and lifetime is fully under the programmer’s control through explicit ALLOCATE and DEALLO-  
 26 CATE statements. The declarations

27 `SUBROUTINE X (N, A, B)`

28 `REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)`

29 specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate  
 30 storage mechanism for the implementation of automatic arrays, but a heap will be needed for some  
 31 allocatable variables.

#### 32 **C.13.1.5 Array constructors (4.7)**

33 Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

34 `(/ 1.0, 3.0, 7.2 /)`

35 which is a rank-one array of size 3,

36 `(/ (1.3, 2.7, L = 1, 10), 7.1 /)`

1 which is a rank-one array of size 21 and contains the pair of real constants 1.3 and 2.7 repeated 10 times  
 2 followed by 7.1, and

3 (/ (I, I = 1, N) /)

4 which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but higher  
 5 dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

## 6 C.13.2 Examples

7 The array features have the potential to simplify the way that almost any array-using program is con-  
 8 ceived and written. Many algorithms involving arrays can now be written conveniently as a series of  
 9 computations with whole arrays.

### 10 C.13.2.1 Unconditional array computations

11 At the simplest level, statements such as

12 A = B + C

13 or

14 S = SUM (A)

15 can take the place of entire DO loops. The loops were required to perform array addition or to sum all  
 16 the elements of an array.

17 Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply	P = MATMUL (Q, R)
largest array element	L = MAXVAL (P)
factorial N	F = PRODUCT ((/ (K, K = 2, N) /))

18 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  
 19 use of the element-by-element definition of array expressions as described in Clause 7. Thus, we can  
 20 write

21 F = SUM (A \* COS (X))

22 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)) /)

23 The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the  
 24 processor is dealing with arrays at every step of the calculation.

### 25 C.13.2.2 Conditional array computations

26 Suppose we wish to compute the Fourier sum in the above example, but to include only those terms  
 27  $a(i) \cos x(i)$  that satisfy the condition that the coefficient  $a(i)$  is less than 0.01 in absolute value. More  
 28 precisely, we are now interested in evaluating the conditional Fourier sum

$$CF = \sum_{|a_i| < 0.01} a_i \times \cos x_i$$

1 where the index runs from 1 to N as before.

2 This can be done by using the MASK parameter of the SUM function, which restricts the summation  
 3 of the elements of the array A \* COS (X) to those elements that correspond to true elements of MASK.  
 4 Clearly, the mask required is the logical array expression ABS (A) < 0.01. Note that the stages of  
 5 evaluation of this expression are:

$$\begin{aligned} A &= (/ A (1), ..., A (N) /) \\ \text{ABS } (A) &= (/ \text{ABS } (A (1)), ..., \text{ABS } (A (N)) /) \\ \text{ABS } (A) < 0.01 &= (/ \text{ABS } (A (1)) < 0.01, ..., \text{ABS } (A (N)) < 0.01 /) \end{aligned}$$

6 The conditional Fourier sum we arrive at is:

7 CF = SUM (A \* COS (X), MASK = ABS (A) < 0.01)

8 If the mask is all false, the value of CF is zero.

9 The use of a mask to define a subset of an array is crucial to the action of the WHERE statement. Thus  
 10 for example, to zero an entire array, we may write simply A = 0; but to set only the negative elements  
 11 to zero, we need to write the conditional assignment

12 WHERE (A .LT. 0) A = 0

13 The WHERE statement complements ordinary array assignment by providing array assignment to any  
 14 subset of an array that can be restricted by a logical expression.

15 In the Ising model described below, the WHERE statement predominates in use over the ordinary array  
 16 assignment statement.

### 17 C.13.2.3 A simple program: the Ising model

18 The Ising model is a well-known Monte Carlo simulation in 3-dimensional Euclidean space which is  
 19 useful in certain physical studies. We will consider in some detail how this might be programmed. The  
 20 model may be described in terms of a logical array of shape N by N by N. Each gridpoint is a single  
 21 logical variable which is to be interpreted as either an up-spin (true) or a down-spin (false).

22 The Ising model operates by passing through many successive states. The transition to the next state is  
 23 governed by a local probabilistic process. At each transition, all gridpoints change state simultaneously.  
 24 Every spin either flips to its opposite state or not according to a rule that depends only on the states  
 25 of its 6 nearest neighbors in the surrounding grid. The neighbors of gridpoints on the boundary faces of  
 26 the model cube are defined by assuming cubic periodicity. In effect, this extends the grid periodically  
 27 by replicating it in all directions throughout space.

28 The rule states that a spin is flipped to its opposite parity for certain gridpoints where a mere 3 or  
 29 fewer of the 6 nearest neighbors have the same parity as it does. Also, the flip is executed only with  
 30 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  
 31 rule seems to promote neighborhood alignments that may presumably lead to equilibrium in the long  
 32 run.)

### 1 C.13.2.3.1 Problems to be solved

2 Some of the programming problems that we will need to solve in order to translate the Ising model into  
3 Fortran statements using entire arrays are

- 4 (1) counting nearest neighbors that have the same spin,
- 5 (2) providing an array function to return an array of random numbers, and
- 6 (3) determining which gridpoints are to be flipped.

### 7 C.13.2.3.2 Solutions in Fortran

8 The arrays needed are:

9 LOGICAL ISING (N, N, N), FLIPS (N, N, N)

10 INTEGER ONES (N, N, N), COUNT (N, N, N)

11 REAL THRESHOLD (N, N, N)

12 The array function needed is:

13 FUNCTION RAND (N)

14 REAL RAND (N, N, N)

15 The transition probabilities are specified in the array

16 REAL P (6)

17 The first task is to count the number of nearest neighbors of each gridpoint  $g$  that have the same spin  
18 as  $g$ .

19 Assuming that ISING is given to us, the statements

20 ONES = 0

21 WHERE (ISING) ONES = 1

22 make the array ONES into an exact analog of ISING in which 1 stands for an up-spin and 0 for a  
23 down-spin.

24 The next array we construct, COUNT, will record for every gridpoint of ISING the number of spins to  
25 be found among the 6 nearest neighbors of that gridpoint. COUNT will be computed by adding together  
26 6 arrays, one for each of the 6 relative positions in which a nearest neighbor is found. Each of the 6  
27 arrays is obtained from the ONES array by shifting the ONES array one place circularly along one of  
28 its dimensions. This use of circular shifting imparts the cubic periodicity.

29 COUNT = CSHIFT (ONES, SHIFT = -1, DIM = 1) &

30 + CSHIFT (ONES, SHIFT = 1, DIM = 1) &

31 + CSHIFT (ONES, SHIFT = -1, DIM = 2) &

32 + CSHIFT (ONES, SHIFT = 1, DIM = 2) &

33 + CSHIFT (ONES, SHIFT = -1, DIM = 3) &

34 + CSHIFT (ONES, SHIFT = 1, DIM = 3)

35 At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints where  
36 the Ising model has a down-spin. But we want a count of down-spins at those gridpoints, so we correct  
37 COUNT at the down (false) points of ISING by writing:



1 WHERE (.NOT. ISING) COUNT = 6 - COUNT

2 Our object now is to use these counts of what may be called the “like-minded nearest neighbors” to  
 3 decide which gridpoints are to be flipped. This decision will be recorded as the true elements of an array  
 4 FLIPS. The decision to flip will be based on the use of uniformly distributed random numbers from the  
 5 interval  $0 \leq p < 1$ . These will be provided at each gridpoint by the array function RAND. The flip will  
 6 occur at a given point if and only if the random number at that point is less than a certain threshold  
 7 value. In particular, by making the threshold value equal to 1 at the points where there are 3 or fewer  
 8 like-minded nearest neighbors, we guarantee that a flip occurs at those points (because  $p$  is always less  
 9 than 1). Similarly, the threshold values corresponding to counts of 4, 5, and 6 are assigned  $P(4)$ ,  $P(5)$ ,  
 10 and  $P(6)$  in order to achieve the desired probabilities of a flip at those points ( $P(4)$ ,  $P(5)$ , and  $P(6)$ )  
 11 are input parameters in the range 0 to 1).

12 The thresholds are established by the statements:

13 THRESHOLD = 1.0

14 WHERE (COUNT == 4) THRESHOLD = P (4)

15 WHERE (COUNT == 5) THRESHOLD = P (5)

16 WHERE (COUNT == 6) THRESHOLD = P (6)

17 and the spins that are to be flipped are located by the statement:

18 FLIPS = RAND (N) <= THRESHOLD

19 All that remains to complete one transition to the next state of the ISING model is to reverse the spins  
 20 in ISING wherever FLIPS is true:

21 WHERE (FLIPS) ISING = .NOT. ISING

### 22 C.13.2.3.3 The complete Fortran subroutine

23 The complete code, enclosed in a subroutine that performs a sequence of transitions, is as follows:

24 SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)

25

26 LOGICAL ISING (N, N, N), FLIPS (N, N, N)

27 INTEGER ONES (N, N, N), COUNT (N, N, N)

28 REAL THRESHOLD (N, N, N), P (6)

29

30 DO I = 1, ITERATIONS

31 ONES = 0

32 WHERE (ISING) ONES = 1

33 COUNT = CSHIFT (ONES, -1, 1) + CSHIFT (ONES, 1, 1) &

34 + CSHIFT (ONES, -1, 2) + CSHIFT (ONES, 1, 2) &

35 + CSHIFT (ONES, -1, 3) + CSHIFT (ONES, 1, 3)

36 WHERE (.NOT. ISING) COUNT = 6 - COUNT

37 THRESHOLD = 1.0

38 WHERE (COUNT == 4) THRESHOLD = P (4)

39 WHERE (COUNT == 5) THRESHOLD = P (5)

40 WHERE (COUNT == 6) THRESHOLD = P (6)

```

1      FLIPS = RAND (N) <= THRESHOLD
2      WHERE (FLIPS) ISING = .NOT. ISING
3  END DO
4
5  CONTAINS
6      FUNCTION RAND (N)
7          REAL RAND (N, N, N)
8          CALL RANDOM_NUMBER (HARVEST = RAND)
9          RETURN
10     END FUNCTION RAND
11 END

```

#### 12 C.13.2.3.4 Reduction of storage

13 The array ISING could be removed (at some loss of clarity) by representing the model in ONES all the  
 14 time. The array FLIPS can be avoided by combining the two statements that use it as:

```

15 WHERE (RAND (N) <= THRESHOLD) ISING = .NOT. ISING

```

16 but an extra temporary array would probably be needed. Thus, the scope for saving storage while  
 17 performing whole array operations is limited. If N is small, this will not matter and the use of whole  
 18 array operations is likely to lead to good execution speed. If N is large, storage may be very important  
 19 and adequate efficiency will probably be available by performing the operations plane by plane. The  
 20 resulting code is not as elegant, but all the arrays except ISING will have size of order  $N^2$  instead of  $N^3$ .

### 21 C.13.3 FORMula TRANslation and array processing

22 Many mathematical formulas can be translated directly into Fortran by use of the array processing  
 23 features.

24 We assume the following array declarations:

```

25 REAL X (N), A (M, N)

```

26 Some examples of mathematical formulas and corresponding Fortran expressions follow.

#### 27 C.13.3.1 A sum of products

The expression

$$\sum_{j=1}^N \prod_{i=1}^M a_{ij}$$

28 can be formed using the Fortran expression

```

29 SUM (PRODUCT (A, DIM=1))

```

30 The argument DIM=1 means that the product is to be computed down each column of A. If A had the  
 31 value  $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$  the result of this expression is  $BE + CF + DG$ .

### 1 C.13.3.2 A product of sums

The expression

$$\prod_{i=1}^M \sum_{j=1}^N a_{ij}$$

2 can be formed using the Fortran expression

3 `PRODUCT (SUM (A, DIM = 2))`

4 The argument `DIM = 2` means that the sum is to be computed along each row of A. If A had the  
5 value  $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$  the result of this expression is  $(B+C+D)(E+F+G)$ .

### 6 C.13.3.3 Addition of selected elements

The expression

$$\sum_{x_i > 0.0} x_i$$

7 can be formed using the Fortran expression

8 `SUM (X, MASK = X > 0.0)`

9 The mask locates the positive elements of the array of rank one. If X has the vector value  $(0.0, -0.1,$   
10  $0.2, 0.3, 0.2, -0.1, 0.0)$ , the result of this expression is 0.7.

### 11 C.13.4 Sum of squared residuals

The expression

$$\sum_{i=1}^N (x_i - x_{\text{mean}})^2$$

12 can be formed using the Fortran statements

13 `XMEAN = SUM (X) / SIZE (X)`

14 `SS = SUM ((X - XMEAN) ** 2)`

15 Thus, SS is the sum of the squared residuals.

### 16 C.13.5 Vector norms: infinity-norm and one-norm

17 The infinity-norm of vector  $X = (X(1), \dots, X(N))$  is defined as the largest of the numbers  $\text{ABS}(X(1)),$   
18  $\dots, \text{ABS}(X(N))$  and therefore has the value  $\text{MAXVAL}(\text{ABS}(X))$ .

19 The one-norm of vector X is defined as the *sum* of the numbers  $\text{ABS}(X(1)), \dots, \text{ABS}(X(N))$  and  
20 therefore has the value  $\text{SUM}(\text{ABS}(X))$ .

### 21 C.13.6 Matrix norms: infinity-norm and one-norm

22 The infinity-norm of the matrix  $A = (A(I, J))$  is the largest row-sum of the matrix  $\text{ABS}(A(I, J))$  and  
23 therefore has the value  $\text{MAXVAL}(\text{SUM}(\text{ABS}(A), \text{DIM} = 2))$ .

24 The one-norm of the matrix  $A = (A(I, J))$  is the largest column-sum of the matrix  $\text{ABS}(A(I, J))$  and  
25 therefore has the value  $\text{MAXVAL}(\text{SUM}(\text{ABS}(A), \text{DIM} = 1))$ .

### 1 C.13.7 Logical queries

2 The intrinsic functions allow quite complicated questions about tabular data to be answered without  
 3 use of loops or conditional constructs. Consider, for example, the questions asked below about a simple  
 4 tabulation of students' test scores.

5 Suppose the rectangular table  $T$  ( $M, N$ ) contains the test scores of  $M$  students who have taken  $N$  different  
 6 tests.  $T$  is an integer matrix with entries in the range 0 to 100.

7 Example: The scores on 4 tests made by 3 students are held as the table

$$T = \begin{bmatrix} 85 & 76 & 90 & 60 \\ 71 & 45 & 50 & 80 \\ 66 & 45 & 21 & 55 \end{bmatrix}$$

8 Question: What is each student's top score?

9 Answer:  $\text{MAXVAL}(T, \text{DIM} = 2)$ ; in the example: [90, 80, 66].

10 Question: What is the average of all the scores?

11 Answer:  $\text{SUM}(T) / \text{SIZE}(T)$ ; in the example: 62.

12 Question: How many of the scores in the table are above average?

13 Answer:  $\text{ABOVE} = T > \text{SUM}(T) / \text{SIZE}(T)$ ;  $N = \text{COUNT}(\text{ABOVE})$ ; in the example: ABOVE is the  
 14 logical array ( $t = \text{true}$ ,  $\cdot = \text{false}$ ):  $\begin{bmatrix} t & t & t & \cdot \\ t & \cdot & \cdot & t \\ t & \cdot & \cdot & \cdot \end{bmatrix}$  and  $\text{COUNT}(\text{ABOVE})$  is 6.

15 Question: What was the lowest score in the above-average group of scores?

16 Answer:  $\text{MINVAL}(T, \text{MASK} = \text{ABOVE})$ , where ABOVE is as defined previously; in the example: 66.

17 Question: Was there a student whose scores were all above average?

18 Answer: With ABOVE as previously defined, the answer is yes or no according as the value of the  
 19 expression  $\text{ANY}(\text{ALL}(\text{ABOVE}, \text{DIM} = 2))$  is true or false; in the example, the answer is no.

### 20 C.13.8 Parallel computations

21 The most straightforward kind of parallel processing is to do the same thing at the same time to many  
 22 operands. Matrix addition is a good example of this very simple form of parallel processing. Thus, the  
 23 array assignment  $A = B + C$  specifies that corresponding elements of the identically-shaped arrays  $B$   
 24 and  $C$  be added together in parallel and that the resulting sums be assigned in parallel to the array  $A$ .

25 The process being done in parallel in the example of matrix addition is of course the process of addi-  
 26 tion; the array feature that implements matrix addition as a parallel process is the element-by-element  
 27 evaluation of array expressions.

28 These observations lead us to look to element-by-element computation as a means of implementing other  
 29 simple parallel processing algorithms.

### 1 C.13.9 Example of element-by-element computation

2 Several polynomials of the same degree may be evaluated at the same point by arranging their coefficients  
 3 as the rows of a matrix and applying Horner's method for polynomial evaluation to the columns of the  
 4 matrix so formed.

5 The procedure is illustrated by the code to evaluate the three cubic polynomials

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$

$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$

$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

6 in parallel at the point  $t = X$  and to place the resulting vector of numbers  $[P(X), Q(X), R(X)]$  in the  
 7 real array RESULT (3).

8 The code to compute RESULT is just the one statement

9 `RESULT = M(:, 1) + X * (M(:, 2) + X * (M(:, 3) + X * M(:, 4)))`

10 where M represents the matrix M (3, 4) with value  $\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$ .

### 11 C.13.10 Bit manipulation and inquiry procedures

12 The procedures IOR, IAND, NOT, IEOR, ISHFT, ISHFTC, IBITS, MVBITS, BTEST, IBSET, and  
 13 IBCLR are defined by MIL-STD 1753 for scalar arguments and are extended in this standard to accept  
 14 array arguments and to return array results.



## Annex D

(Informative)

## Syntax rules

### D.1 Extract of all syntax rules

## Section 1:

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

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

R103    *scalar-xyz*                      is    *xyz*

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

## Section 2:

[illegible]

R202	<i>program-unit</i>	<b>is</b>	<i>main-program</i>
		<b>or</b>	<i>external-subprogram</i>
		<b>or</b>	<i>module</i>
		<b>or</b>	<i>submodule</i>
		<b>or</b>	<i>block-data</i>

[illegible][illegible][illegible]

R206	<i>implicit-part-stmt</i>	<b>is</b>	<i>implicit-stmt</i>
		<b>or</b>	<i>parameter-stmt</i>
		<b>or</b>	<i>format-stmt</i>
		<b>or</b>	<i>entry-stmt</i>

R207	<i>declaration-construct</i>	<b>is</b> <i>derived-type-def</i>
		<b>or</b> <i>entry-stmt</i>
		<b>or</b> <i>enum-def</i>
		<b>or</b> <i>format-stmt</i>
		<b>or</b> <i>interface-block</i>
		<b>or</b> <i>macro-definition</i>
		<b>or</b> <i>parameter-stmt</i>
		<b>or</b> <i>procedure-declaration-stmt</i>
		<b>or</b> <i>specification-stmt</i>
		<b>or</b> <i>type-declaration-stmt</i>
		<b>or</b> <i>stmt-function-stmt</i>

R208	<i>execution-part</i>	<b>is</b> <i>executable-construct</i> [ <i>execution-part-construct</i> ] ...
R209	<i>execution-part-construct</i>	<b>is</b> <i>executable-construct</i> <b>or</b> <i>format-stmt</i> <b>or</b> <i>entry-stmt</i> <b>or</b> <i>data-stmt</i>
R210	<i>internal-subprogram-part</i>	<b>is</b> <i>contains-stmt</i> [ <i>internal-subprogram</i> ] ...
R211	<i>internal-subprogram</i>	<b>is</b> <i>function-subprogram</i> <b>or</b> <i>subroutine-subprogram</i>
R212	<i>specification-stmt</i>	<b>is</b> <i>access-stmt</i> <b>or</b> <i>allocatable-stmt</i> <b>or</b> <i>asynchronous-stmt</i> <b>or</b> <i>bind-stmt</i> <b>or</b> <i>common-stmt</i> <b>or</b> <i>data-stmt</i> <b>or</b> <i>dimension-stmt</i> <b>or</b> <i>equivalence-stmt</i> <b>or</b> <i>external-stmt</i> <b>or</b> <i>intent-stmt</i> <b>or</b> <i>intrinsic-stmt</i> <b>or</b> <i>namelist-stmt</i> <b>or</b> <i>optional-stmt</i> <b>or</b> <i>pointer-stmt</i> <b>or</b> <i>protected-stmt</i> <b>or</b> <i>save-stmt</i> <b>or</b> <i>target-stmt</i> <b>or</b> <i>volatile-stmt</i> <b>or</b> <i>value-stmt</i>
R213	<i>executable-construct</i>	<b>is</b> <i>action-stmt</i> <b>or</b> <i>associate-construct</i> <b>or</b> <i>block-construct</i> <b>or</b> <i>case-construct</i> <b>or</b> <i>critical-construct</i> <b>or</b> <i>do-construct</i> <b>or</b> <i>forall-construct</i> <b>or</b> <i>if-construct</i> <b>or</b> <i>select-type-construct</i> <b>or</b> <i>where-construct</i>
R214	<i>action-stmt</i>	<b>is</b> <i>allocate-stmt</i> <b>or</b> <i>assignment-stmt</i> <b>or</b> <i>backspace-stmt</i> <b>or</b> <i>call-stmt</i> <b>or</b> <i>close-stmt</i> <b>or</b> <i>continue-stmt</i> <b>or</b> <i>cycle-stmt</i>



**or** *deallocate-stmt*  
**or** *endfile-stmt*  
**or** *end-function-stmt*  
**or** *end-program-stmt*  
**or** *end-subroutine-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-program-stmt*, or *end-subroutine-stmt*.

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

### Section 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** *real-literal-constant*

		<b>or</b> <i>complex-literal-constant</i>
		<b>or</b> <i>logical-literal-constant</i>
		<b>or</b> <i>char-literal-constant</i>
		<b>or</b> <i>boz-literal-constant</i>
R307	<i>named-constant</i>	<b>is</b> <i>name</i>
R308	<i>int-constant</i>	<b>is</b> <i>constant</i>
C302	(R308) <i>int-constant</i> shall be of type integer.	
R309	<i>char-constant</i>	<b>is</b> <i>constant</i>
C303	(R309) <i>char-constant</i> shall be of type character.	
R310	<i>intrinsic-operator</i>	<b>is</b> <i>power-op</i>
		<b>or</b> <i>mult-op</i>
		<b>or</b> <i>add-op</i>
		<b>or</b> <i>concat-op</i>
		<b>or</b> <i>rel-op</i>
		<b>or</b> <i>not-op</i>
		<b>or</b> <i>and-op</i>
		<b>or</b> <i>or-op</i>
		<b>or</b> <i>equiv-op</i>
R311	<i>defined-operator</i>	<b>is</b> <i>defined-unary-op</i>
		<b>or</b> <i>defined-binary-op</i>
		<b>or</b> <i>extended-intrinsic-op</i>
R312	<i>extended-intrinsic-op</i>	<b>is</b> <i>intrinsic-operator</i>
R313	<i>label</i>	<b>is</b> <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>	<b>is</b> <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>	<b>is</b> DEFINE MACRO [ , <i>macro-attribute-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-attribute</i>	<b>is</b> <i>access-spec</i>
R317	<i>macro-declaration-stmt</i>	<b>is</b> <i>macro-type-declaration-stmt</i>
		<b>or</b> <i>macro-optional-decl-stmt</i>
R318	<i>macro-type-declaration-stmt</i>	<b>is</b> MACRO <i>macro-type-spec</i> :: <i>macro-local-variable-name-list</i>
R319	<i>macro-optional-decl-stmt</i>	<b>is</b> MACRO OPTIONAL :: <i>macro-dummy-arg-name-list</i>
R320	<i>macro-type-spec</i>	<b>is</b> INTEGER [ ( [ KIND= ] <i>macro-expr</i> ) ]
C306	(R318) 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	(R320) If <i>macro-expr</i> appears, when the macro is expanded it shall be of type integer, and have a non-negative value that specifies a representation method that exists on the processor.	
R321	<i>macro-body-block</i>	<b>is</b> [ <i>macro-body-construct</i> ] ...
R322	<i>macro-body-construct</i>	<b>is</b> <i>macro-definition</i>
		<b>or</b> <i>expand-stmt</i>

- or** *macro-body-stmt*  
**or** *macro-do-construct*  
**or** *macro-if-construct*
- 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.
- R323 *macro-do-construct* **is** *macro-do-stmt*  
*macro-body-block*  
*macro-end-do-stmt*
- R324 *macro-do-stmt* **is** MACRO DO *macro-do-variable-name* = *macro-do-limit* , ■  
■ *macro-do-limit* [ , *macro-do-limit* ]
- C310 (R324) A *macro-do-variable-name* shall be a local variable of the macro being defined, and shall not be a macro dummy argument.
- R325 *macro-do-limit* **is** *macro-expr*
- C311 (R325) A *macro-do-limit* shall expand to an expression of type integer.
- R326 *macro-end-do-stmt* **is** MACRO END DO
- R327 *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*
- R328 *macro-if-then-stmt* **is** MACRO IF ( *macro-condition* ) THEN
- R329 *macro-else-if-stmt* **is** MACRO ELSE IF ( *macro-condition* ) THEN
- R330 *macro-else-stmt* **is** MACRO ELSE
- R331 *macro-end-if-stmt* **is** MACRO END IF
- R332 *macro-condition* **is** *macro-expr*
- C312 (R332) A macro condition shall expand to an expression of type logical.
- R333 *macro-body-stmt* **is** *result-token* [ *result-token* ] ... [ && ]
- C313 (R333) The first *result-token* shall not be MACRO unless the second *result-token* is not a keyword or name.
- R334 *result-token* **is** *token* [ %% *token* ] ...
- C314 (R334) The concatenated textual *tokens* in a *result-token* shall have the form of a lexical token.
- R335 *token* **is** any lexical token including labels, keywords, and semi-colon.
- C315 && shall not appear in the last *macro-body-stmt* of a macro definition.
- C316 When a macro is expanded, the last *macro-body-stmt* processed shall not end with &&.
- R336 *end-macro-stmt* **is** END MACRO [ *macro-name* ]
- C317 (R314) The *macro-name* in the END MACRO statement shall be the same as the *macro-name* in the DEFINE MACRO statement.
- R337 *macro-expr* **is** *basic-token-sequence*
- C318 (R337) A *macro-expr* shall expand to a scalar initialization expression.
- R338 *expand-stmt* **is** EXPAND *macro-name* [ ( *macro-actual-arg-list* ) ]
- C319 (R338) *macro-name* shall be the name of a macro that was previously defined or accessed via use or host association.
- C320 (R338) The macro shall expand to a sequence or zero or more complete Fortran statements.
- C321 (R338) The statements produced by a macro expansion shall conform to the syntax rules and

- constraints as if they physically replaced the EXPAND statement prior to program processing.
- C322 (R338) The statements produced by a macro expansion shall not include a statement which ends the scoping unit containing the EXPAND statement.
- C323 (R338) If a macro expansion produces a statement which begins a new scoping unit, it shall also produce a statement which ends that scoping unit.
- C324 (R338) 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 *if-stmt*, *end-program-stmt*, *end-function-stmt*, or *end-subroutine-stmt*.
- C325 (R338) If the EXPAND statement appears as a *do-term-action-stmt*, it shall expand to exactly one *action-stmt* that is not a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.
- C326 (R338) 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.
- C327 (R338) A *macro-actual-arg* shall appear corresponding to each nonoptional macro dummy argument.
- C328 (R338) At most one *macro-actual-arg* shall appear corresponding to each optional macro dummy argument.
- R339 *macro-actual-arg*                    **is** [ *macro-dummy-name* = ] *macro-actual-arg-value*
- C329 (R339) *macro-dummy-name* shall be the name of a macro dummy argument of the macro being expanded.
- C330 (R338) The *macro-dummy-name*= shall not be omitted unless it has been omitted from each preceding *macro-actual-arg* in the *expand-stmt*.
- R340 *macro-actual-arg-value*        **is** basic-token-sequence
- R341 *basic-token-sequence*           **is** *basic-token*  
                                          **or** [ *basic-token-sequence* ] *nested-token-sequence* ■  
                                          ■ [ *basic-token-sequence* ]  
                                          **or** *basic-token basic-token-sequence*
- R342 *basic-token*                    **is** any lexical token except comma, parentheses, array ■  
                                          ■ constructor delimiters, and semi-colon.
- R343 *nested-token-sequence*        **is** ( [ *arg-token* ] ... )  
                                          **or** ( / [ *arg-token* ] ... / )  
                                          **or** *lbracket* [ *arg-token* ] ... *rbracket*
- R344 *arg-token*                      **is** *basic-token*  
                                          **or** ,

#### Section 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 may be used as a *type-param-value* only 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.
- 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** *scalar-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 *scalar-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, 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 *boz-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.
- R433 *private-or-sequence* **is** *private-components-stmt*  
**or** *sequence-stmt*
- C434 (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* ]
- C435 (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
- C436 (R439) If SEQUENCE appears, each data component shall be declared to be of an intrinsic type or of a sequence derived type.
- C437 (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* ]
- C438 (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*.
- C439 (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 *component-dim-spec*  
or CONTIGUOUS  
or POINTER
- R443 *component-decl* is *component-name* [ ( *component-array-spec* ) ] ■  
■ [ *lbracket co-array-spec rbracket* ] ■  
■ [ \* *char-length* ] [ *component-initialization* ]
- R444 *component-dim-spec* is DIMENSION ( *component-array-spec* )  
or DIMENSION [ ( *deferred-shape-spec-list* ) ] ■  
■ *lbracket co-array-spec rbracket*
- R445 *component-array-spec* is *explicit-shape-spec-list*  
or *deferred-shape-spec-list*
- C440 (R441) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.
- C441 (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.
- C442 (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.
- C443 (R441) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.
- C444 (R441) If a *co-array-spec* appears, it shall be a *deferred-shape-spec-list* and the component shall have the ALLOCATABLE attribute.
- C445 A data component whose type has a co-array ultimate component shall be a nonpointer nonallocatable scalar and shall not be a co-array.
- C446 (R441) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.
- C447 (R445) 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.
- C448 (R441) A component shall not have both the ALLOCATABLE and the POINTER attribute.
- C449 (R441) If the CONTIGUOUS attribute is specified, the component shall be an array with the POINTER attribute.
- C450 (R443) The \* *char-length* option is permitted only if the component is of type character.
- C451 (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.
- R446 *proc-component-def-stmt* is PROCEDURE ( [ *proc-interface* ] ) , ■  
■ *proc-component-attr-spec-list* :: *proc-decl-list*
- R447 *proc-component-attr-spec* is POINTER  
or PASS [ ( *arg-name* ) ]  
or NOPASS



**or** *access-spec*

- C452 (R446) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-component-def-stmt*.
- C453 (R446) **POINTER** shall appear in each *proc-component-attr-spec-list*.
- C454 (R446) If the procedure pointer component has an implicit interface or has no arguments, **NOPASS** shall be specified.
- C455 (R446) If **PASS** (*arg-name*) appears, the interface shall have a dummy argument named *arg-name*.
- C456 (R446) **PASS** and **NOPASS** shall not both appear in the same *proc-component-attr-spec-list*.
- C457 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.
- R448 *component-initialization*      **is** = *initialization-expr*  
                                          **or** => *null-init*  
                                          **or** => *initial-data-target*
- R449 *initial-data-target*      **is** *designator*
- C458 (R441) If *component-initialization* appears, a double-colon separator shall appear before the *component-decl-list*.
- C459 (R441) If *component-initialization* appears, every type parameter and array bound of the component shall be an initialization expression.
- C460 (R441) If => appears in *component-initialization*, **POINTER** shall appear in the *component-attr-spec-list*. If = appears in *component-initialization*, **POINTER** or **ALLOCATABLE** shall not appear in the *component-attr-spec-list*.
- C461 (R448) If *initial-data-target* appears, *component-name* shall be data-pointer-initialization compatible with it.
- C462 (R449) 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.
- R450 *private-components-stmt*      **is** **PRIVATE**
- C463 (R450) A *private-components-stmt* is permitted only if the type definition is within the specification part of a module.
- R451 *type-bound-procedure-part*      **is** *contains-stmt*  
                                          [ *binding-private-stmt* ]  
                                          [ *proc-binding-stmt* ] ...
- R452 *binding-private-stmt*      **is** **PRIVATE**
- C464 (R451) A *binding-private-stmt* is permitted only if the type definition is within the specification part of a module.
- R453 *proc-binding-stmt*      **is** *specific-binding*  
                                          **or** *generic-binding*  
                                          **or** *final-binding*
- R454 *specific-binding*      **is** **PROCEDURE** [ (*interface-name*) ] ■  
                                          ■ [ [ , *binding-attr-list* ] :: ] ■  
                                          ■ *binding-name* [ => *procedure-name* ]
- C465 (R454) If => *procedure-name* appears, the double-colon separator shall appear.
- C466 (R454) If => *procedure-name* appears, *interface-name* shall not appear.
- C467 (R454) The *procedure-name* shall be the name of an accessible module procedure or an external procedure that has an explicit interface.
- R455 *generic-binding*      **is** **GENERIC** ■

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

- C468 (R455) Within the *specification-part* of a module, each *generic-binding* shall specify, either implicitly or explicitly, the same accessibility as every other *generic-binding* with that *generic-spec* in the same derived type.
- C469 (R455) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the type.
- C470 (R455) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object dummy argument (4.5.4.4).
- C471 (R455) If *generic-spec* is OPERATOR ( *defined-operator* ), the interface of each binding shall be as specified in 12.4.2.1.1.
- C472 (R455) If *generic-spec* is ASSIGNMENT ( = ), the interface of each binding shall be as specified in 12.4.2.1.2.
- C473 (R455) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in 9.5.3.7. The type of the *dtv* argument shall be *type-name*.
- R456 *binding-attr*                    **is** PASS [ ( *arg-name* ) ]  
                                      **or** NOPASS  
                                      **or** NON\_OVERRIDABLE  
                                      **or** DEFERRED  
                                      **or** *access-spec*
- C474 (R456) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.
- C475 (R454) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.
- C476 (R454) If PASS ( *arg-name* ) appears, the interface of the binding shall have a dummy argument named *arg-name*.
- C477 (R456) PASS and NOPASS shall not both appear in the same *binding-attr-list*.
- C478 (R456) NON\_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-attr-list*.
- C479 (R456) DEFERRED shall appear if and only if *interface-name* appears.
- C480 (R454) An overriding binding (4.5.7.3) shall have the DEFERRED attribute only if the binding it overrides is deferred.
- C481 (R454) A binding shall not override an inherited binding (4.5.7.2) that has the NON\_OVERRIDABLE attribute.
- R457 *final-binding*                    **is** FINAL [ :: ] *final-subroutine-name-list*
- C482 (R457) 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 be INTENT(OUT).
- C483 (R457) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.
- C484 (R457) 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.
- R458 *derived-type-spec*                **is** *type-name* [ ( *type-param-spec-list* ) ]
- R459 *type-param-spec*                **is** [ *keyword* = ] *type-param-value*
- C485 (R458) *type-name* shall be the name of an accessible derived type.
- C486 (R458) *type-param-spec-list* shall appear only if the type is parameterized.
- C487 (R458) 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.
- C488 (R459) 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*.
- C489 (R459) Each *keyword* shall be the name of a parameter of the type.
- C490 (R459) 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.
- R460 *structure-constructor*      **is**   *derived-type-spec* ( [ *component-spec-list* ] )
- R461 *component-spec*              **is**   [ *keyword* = ] *component-data-source*
- R462 *component-data-source*      **is**   *expr*  
                                          **or**   *data-target*  
                                          **or**   *proc-target*
- C491 (R460) The *derived-type-spec* shall not specify an abstract type (4.5.7).
- C492 (R460) At most one *component-spec* shall be provided for a component.
- C493 (R460) 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.
- C494 (R460) 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.
- C495 (R461) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been omitted from each preceding *component-spec* in the constructor.
- C496 (R461) Each *keyword* shall be the name of a component of the type.
- C497 (R460) 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.
- C498 (R460) 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 (12.5.4.1).
- C499 (R462) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall correspond to a procedure pointer component.
- C4100 (R462) A *data-target* shall have the same rank as its corresponding component.
- R463 *enum-def*                      **is**   *enum-def-stmt*  
                                          *enumerator-def-stmt*  
                                          [ *enumerator-def-stmt* ] ...  
                                          *end-enum-stmt*
- R464 *enum-def-stmt*                **is**   ENUM, BIND(C)
- R465 *enumerator-def-stmt*        **is**   ENUMERATOR [ :: ] *enumerator-list*
- R466 *enumerator*                   **is**   *named-constant* [ = *scalar-int-initialization-expr* ]
- R467 *end-enum-stmt*               **is**   END ENUM
- C4101 (R465) If = appears in an *enumerator*, a double-colon separator shall appear before the *enumerator-list*.
- R468 *array-constructor*          **is**   ( / *ac-spec* / )  
                                          **or**   *lbracket ac-spec rbracket*
- R469 *ac-spec*                        **is**   *type-spec* ::  
                                          **or**   [ *type-spec* :: ] *ac-value-list*
- R470 *lbracket*                      **is**   [
- R471 *rbracket*                       **is**   ]
- R472 *ac-value*                       **is**   *expr*  
                                          **or**   *ac-implied-do*
- R473 *ac-implied-do*                **is**   ( *ac-value-list* , *ac-implied-do-control* )
- R474 *ac-implied-do-control*       **is**   *ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■

■ [ , *scalar-int-expr* ]

R475 *ac-do-variable*                    **is** *do-variable*

C4102 (R469) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameters.

C4103 (R469) 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 7.10.

C4104 (R469) 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*.

C4105 (R473) 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*.

## Section 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 entity shall not be explicitly given any attribute more than once in a scoping unit.
- C514 An *array-spec* for a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.
- C515 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
- C516 (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* ])
- C517 An entity with the BIND attribute shall be a common block, variable, or procedure.
- C518 A variable with the BIND attribute shall be declared in the specification part of a module.
- C519 A variable with the BIND attribute shall be interoperable (15.3).
- C520 Each variable of a common block with the BIND attribute shall be interoperable.
- C521 (R508) The *scalar-char-initialization-expr* shall be of default character kind.
- C522 An entity that has 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*
- C523 (R501) A co-array that has the ALLOCATABLE attribute shall be specified with a *co-array-spec* that is a *deferred-shape-spec-list*.
- C524 A co-array shall be a component or a variable that is not a function result.
- C525 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.
- 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-shape-spec-list*  
**or** *assumed-size-spec*
- C526 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*
- C527 (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** :
- C528 An array that has 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-list* , ] [ *lower-bound* : ] \*
- C529 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy data argument.
- C530 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* : ] \*
- C531 An implied-shape array shall be a named constant.
- C532 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.
- R519 *intent-spec* **is** IN  
**or** OUT  
**or** INOUT
- C533 An entity with the INTENT attribute shall be a dummy data object or a dummy procedure pointer.
- C534 (R519) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition context (16.6.7).
- C535 A pointer with the INTENT (IN) attribute shall not appear in a pointer association context (16.6.8).
- C536 If the name of a generic intrinsic procedure is explicitly declared to have the INTRINSIC attribute, and it is also the generic name of one or more generic interfaces (12.4.2.1) accessible in the same scoping unit, the procedures in the interfaces and the specific intrinsic procedures shall all be functions or all be subroutines, and the characteristics of the specific intrinsic procedures and the procedures in the interfaces shall differ as specified in 16.3.4.
- C537 An entity with the OPTIONAL attribute shall be a dummy argument.
- C538 An entity with the PARAMETER attribute shall not be a variable, a co-array, or a procedure.
- C539 An entity with the POINTER attribute shall not have the ALLOCATABLE, INTRINSIC, or TARGET attribute.
- C540 A procedure with the POINTER attribute shall have the EXTERNAL attribute.
- C541 A co-array shall not have the POINTER attribute.
- C542 The PROTECTED attribute shall be specified only in the specification part of a module.
- C543 An entity with the PROTECTED attribute shall be a procedure pointer or variable.
- C544 An entity with the PROTECTED attribute shall not be in a common block.
- C545 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (16.6.7) or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.
- C546 A pointer that has the PROTECTED attribute and is accessed by use association shall not appear in a pointer association context (16.6.8).
- (1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,
  - (2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or
  - (3) An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.



- R533 *data-i-do-variable* **is** *do-variable*
- C560 A *data-stmt-object* or *data-i-do-object* shall not be a co-indexed variable.
- C561 (R530) 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.
- C562 (R530) 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.
- C563 (R530) 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*.
- C564 (R532) The *array-element* shall be a variable.
- C565 (R532) The *scalar-structure-component* shall be a variable.
- C566 (R532) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.
- C567 (R532) 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*.
- R534 *data-stmt-value* **is** [ *data-stmt-repeat* \* ] *data-stmt-constant*
- R535 *data-stmt-repeat* **is** *scalar-int-constant*  
**or** *scalar-int-constant-subobject*
- C568 (R535) 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.
- R536 *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*
- C569 (R536) 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.
- C570 (R536) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.
- R537 *int-constant-subobject* **is** *constant-subobject*
- C571 (R537) *int-constant-subobject* shall be of type integer.
- R538 *constant-subobject* **is** *designator*
- C572 (R538) *constant-subobject* shall be a subobject of a constant.
- C573 (R538) Any subscript, substring starting point, or substring ending point shall be an initialization expression.
- R539 *dimension-stmt* **is** DIMENSION [ :: ] *dimension-decl-list*
- R540 *dimension-decl* **is** *array-name* ( *array-spec* )  
**or** *co-array-name* [ ( *array-spec* ) ] *lbracket co-array-spec rbracket*
- R541 *intent-stmt* **is** INTENT ( *intent-spec* ) [ :: ] *dummy-arg-name-list*
- R542 *optional-stmt* **is** OPTIONAL [ :: ] *dummy-arg-name-list*
- R543 *parameter-stmt* **is** PARAMETER ( *named-constant-def-list* )
- R544 *named-constant-def* **is** *named-constant* = *initialization-expr*



- R545 *pointer-stmt* **is** `POINTER [ :: ] pointer-decl-list`
- R546 *pointer-decl* **is** `object-name [ ( deferred-shape-spec-list ) ]`  
**or** `proc-entity-name`
- R547 *protected-stmt* **is** `PROTECTED [ :: ] entity-name-list`
- R548 *save-stmt* **is** `SAVE [ [ :: ] saved-entity-list ]`
- R549 *saved-entity* **is** `object-name`  
**or** `proc-pointer-name`  
**or** `/ common-block-name /`
- R550 *proc-pointer-name* **is** `name`
- C574 (R548) 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.
- R551 *target-stmt* **is** `TARGET [ :: ] target-decl-list`
- R552 *target-decl* **is** `object-name [ ( array-spec ) ] ■`  
**■** `[ lbracket co-array-spec rbracket ]`
- R553 *value-stmt* **is** `VALUE [ :: ] dummy-arg-name-list`
- R554 *volatile-stmt* **is** `VOLATILE [ :: ] object-name-list`
- R555 *implicit-stmt* **is** `IMPLICIT implicit-spec-list`  
**or** `IMPLICIT NONE`
- R556 *implicit-spec* **is** `declaration-type-spec ( letter-spec-list )`
- R557 *letter-spec* **is** `letter [ – letter ]`
- C575 (R555) 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.
- C576 (R557) If the minus and second *letter* appear, the second letter shall follow the first letter alphabetically.
- R558 *namelist-stmt* **is** `NAMELIST ■`  
**■** `/ namelist-group-name / namelist-group-object-list ■`  
**■** `[ [ , ] / namelist-group-name / ■`  
**■** `namelist-group-object-list ] ...`
- C577 (R558) The *namelist-group-name* shall not be a name accessed by use association.
- R559 *namelist-group-object* **is** `variable-name`
- C578 (R559) A *namelist-group-object* shall not be an assumed-size array.
- C579 (R558) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.
- R560 *equivalence-stmt* **is** `EQUIVALENCE equivalence-set-list`
- R561 *equivalence-set* **is** `( equivalence-object , equivalence-object-list )`
- R562 *equivalence-object* **is** `variable-name`  
**or** `array-element`  
**or** `substring`
- C580 (R562) 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.

- C581 (R562) An *equivalence-object* shall not be a designator that has more than one *part-ref*.
- C582 (R562) An *equivalence-object* shall not be a co-array or a subobject thereof.
- C583 (R562) An *equivalence-object* shall not have the TARGET attribute.
- C584 (R562) Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.7).
- C585 (R561) If an *equivalence-object* is of type default integer, default real, double precision real, default complex, default logical, or numeric sequence type, all of the objects in the equivalence set shall be of these types.
- C586 (R561) 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.
- C587 (R561) 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.
- C588 (R561) 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.
- C589 (R562) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equivalence set shall have the PROTECTED attribute.
- C590 (R562) The name of an *equivalence-object* shall not be a name made accessible by use association.
- C591 (R562) A *substring* shall not have length zero.
- R563 *common-stmt*                    **is**    **COMMON** ■  
                                                  ■ [ / [ *common-block-name* ] / ] *common-block-object-list* ■  
                                                  ■ [ [ , ] / [ *common-block-name* ] / ■  
                                                  ■ *common-block-object-list* ] ...
- R564 *common-block-object*        **is**    *variable-name* [ ( *array-spec* ) ]  
                                                  **or**    *proc-pointer-name*
- C592 (R564) An *array-spec* in a *common-block-object* shall be an *explicit-shape-spec-list*.
- C593 (R564) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all *common-block-object-lists* within a scoping unit.
- C594 (R564) 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.
- C595 (R564) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.2) or a type with the BIND attribute and it shall have no default initialization.
- C596 (R564) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use association.

## Section 6:

- R601 *variable*                    **is**    *designator*  
                                                  **or**    *expr*
- C601 (R601) *designator* shall not be a constant or a subobject of a constant.
- C602 (R601) The *expr* shall be a reference to a function that has a pointer result.
- R602 *variable-name*                **is**    *name*
- C603 (R602) A *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 and *part-name* is an array, *section-subscript-list* shall appear.  
C616 (R613) A *data-ref* that is a co-indexed object shall not be of a type that has a pointer ultimate component.  
C617 (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*  
C618 (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  
C619 (R615) The *designator* shall be of complex type.  
R616 *type-param-inquiry* **is** *designator* % *type-param-name*  
C620 (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*  
C621 (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*  
C622 (R618) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a

*section-subscript-list* with nonzero rank, the *array-section* is a *complex-part-designator* that is an array, or another *part-ref* shall have nonzero rank.

C623 (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*

C624 (R623) A *vector-subscript* shall be an integer array expression of rank one.

C625 (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*

C626 (R632) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

C627 (R626) If any *allocate-object* in the statement has a deferred type parameter, either *type-spec* or SOURCE= shall appear.

C628 (R626) If a *type-spec* appears, it shall specify a type with which each *allocate-object* is type compatible.

C629 (R626) If any *allocate-object* is unlimited polymorphic or is of abstract type, either *type-spec* or SOURCE= shall appear.

C630 (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.

C631 (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*.

C632 (R631) An *allocate-shape-spec-list* shall appear if and only if the *allocate-object* is an array. An *allocate-co-array-spec* shall appear if and only if the *allocate-object* is a co-array.

C633 (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*.

- C634 (R627) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.
- C635 (R626) At most one of SOURCE=, MOLD=, and *type-spec* shall appear.
- C636 (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*.
- C637 (R626) Corresponding kind type parameters of *allocate-object* and *source-expr* shall have the same values.
- C638 (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*
- C639 (R639) Each *pointer-object* shall have the POINTER attribute.
- R640 *deallocate-stmt*               **is** DEALLOCATE ( *allocate-object-list* [ , *dealloc-opt-list* ] )
- C640 (R640) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.
- R641 *dealloc-opt*                   **is** STAT = *stat-variable*  
                                      **or** ERRMSG = *errmsg-variable*
- C641 (R641) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

## Section 7:

- 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.
- 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*.
- 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** -
- R710 *level-3-expr*                   **is** [ *level-3-expr concat-op* ] *level-2-expr*
- R711 *concat-op*                      **is** //
- R712 *level-4-expr*                   **is** [ *level-3-expr rel-op* ] *level-3-expr*
- R713 *rel-op*                         **is** .EQ.  
                                      **or** .NE.

		<b>or</b> .LT.
		<b>or</b> .LE.
		<b>or</b> .GT.
		<b>or</b> .GE.
		<b>or</b> ==
		<b>or</b> /=
		<b>or</b> <
		<b>or</b> <=
		<b>or</b> >
		<b>or</b> >=
R714	<i>and-operand</i>	<b>is</b> [ <i>not-op</i> ] <i>level-4-expr</i>
R715	<i>or-operand</i>	<b>is</b> [ <i>or-operand and-op</i> ] <i>and-operand</i>
R716	<i>equiv-operand</i>	<b>is</b> [ <i>equiv-operand or-op</i> ] <i>or-operand</i>
R717	<i>level-5-expr</i>	<b>is</b> [ <i>level-5-expr equiv-op</i> ] <i>equiv-operand</i>
R718	<i>not-op</i>	<b>is</b> .NOT.
R719	<i>and-op</i>	<b>is</b> .AND.
R720	<i>or-op</i>	<b>is</b> .OR.
R721	<i>equiv-op</i>	<b>is</b> .EQV. <b>or</b> .NEQV. <b>or</b> .XOR.
R722	<i>expr</i>	<b>is</b> [ <i>expr defined-binary-op</i> ] <i>level-5-expr</i>
R723	<i>defined-binary-op</i>	<b>is</b> . <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	<i>logical-expr</i>	<b>is</b> <i>expr</i>
C705	(R724) <i>logical-expr</i> shall be of type logical.	
R725	<i>char-expr</i>	<b>is</b> <i>expr</i>
C706	(R725) <i>char-expr</i> shall be of type character.	
R726	<i>default-char-expr</i>	<b>is</b> <i>expr</i>
C707	(R726) <i>default-char-expr</i> shall be of type default character.	
R727	<i>int-expr</i>	<b>is</b> <i>expr</i>
C708	(R727) <i>int-expr</i> shall be of type integer.	
R728	<i>numeric-expr</i>	<b>is</b> <i>expr</i>
C709	(R728) <i>numeric-expr</i> shall be of type integer, real, or complex.	
C710	The kind type parameter for the result of a bits concatenation operation expression shall be a bits kind type parameter value supported by the processor.	
R729	<i>specification-expr</i>	<b>is</b> <i>scalar-int-expr</i>
C711	(R729) The <i>scalar-int-expr</i> shall be a restricted expression.	
R730	<i>initialization-expr</i>	<b>is</b> <i>expr</i>
C712	(R730) <i>initialization-expr</i> shall be an initialization expression.	
R731	<i>char-initialization-expr</i>	<b>is</b> <i>char-expr</i>
C713	(R731) <i>char-initialization-expr</i> shall be an initialization expression.	
R732	<i>int-initialization-expr</i>	<b>is</b> <i>int-expr</i>
C714	(R732) <i>int-initialization-expr</i> shall be an initialization expression.	
R733	<i>logical-initialization-expr</i>	<b>is</b> <i>logical-expr</i>
C715	(R733) <i>logical-initialization-expr</i> 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, either *data-pointer-object* shall be type compatible (4.3.1.3) with it and the corresponding kind type parameters shall be equal, or *data-target* and *data-pointer-object* shall be bits compatible.
- 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* ] ...

		[ <i>masked-elsewhere-stmt</i>
		[ <i>where-body-construct</i> ] ... ] ...
		[ <i>elsewhere-stmt</i>
		[ <i>where-body-construct</i> ] ... ]
		<i>end-where-stmt</i>
R745	<i>where-construct-stmt</i>	<b>is</b> [ <i>where-construct-name</i> :] WHERE ( <i>mask-expr</i> )
R746	<i>where-body-construct</i>	<b>is</b> <i>where-assignment-stmt</i>
		<b>or</b> <i>where-stmt</i>
		<b>or</b> <i>where-construct</i>
R747	<i>where-assignment-stmt</i>	<b>is</b> <i>assignment-stmt</i>
R748	<i>mask-expr</i>	<b>is</b> <i>logical-expr</i>
R749	<i>masked-elsewhere-stmt</i>	<b>is</b> ELSEWHERE ( <i>mask-expr</i> ) [ <i>where-construct-name</i> ]
R750	<i>elsewhere-stmt</i>	<b>is</b> ELSEWHERE [ <i>where-construct-name</i> ]
R751	<i>end-where-stmt</i>	<b>is</b> END WHERE [ <i>where-construct-name</i> ]
C734	(R747) A <i>where-assignment-stmt</i> that is a defined assignment shall be elemental.	
C735	(R744) If the <i>where-construct-stmt</i> is identified by a <i>where-construct-name</i> , the corresponding <i>end-where-stmt</i> shall specify the same <i>where-construct-name</i> . If the <i>where-construct-stmt</i> is not identified by a <i>where-construct-name</i> , the corresponding <i>end-where-stmt</i> shall not specify a <i>where-construct-name</i> . If an <i>elsewhere-stmt</i> or a <i>masked-elsewhere-stmt</i> is identified by a <i>where-construct-name</i> , the corresponding <i>where-construct-stmt</i> shall specify the same <i>where-construct-name</i> .	
C736	(R746) A statement that is part of a <i>where-body-construct</i> shall not be a branch target statement.	
R752	<i>forall-construct</i>	<b>is</b> <i>forall-construct-stmt</i>
		[ <i>forall-body-construct</i> ] ...
		<i>end-forall-stmt</i>
R753	<i>forall-construct-stmt</i>	<b>is</b> [ <i>forall-construct-name</i> :] FORALL <i>forall-header</i>
R754	<i>forall-header</i>	<b>is</b> ( <i>forall-triplet-spec-list</i> [ , <i>scalar-mask-expr</i> ] )
R755	<i>forall-triplet-spec</i>	<b>is</b> <i>index-name</i> = <i>subscript</i> : <i>subscript</i> [ : <i>stride</i> ]
R756	<i>forall-body-construct</i>	<b>is</b> <i>forall-assignment-stmt</i>
		<b>or</b> <i>where-stmt</i>
		<b>or</b> <i>where-construct</i>
		<b>or</b> <i>forall-construct</i>
		<b>or</b> <i>forall-stmt</i>
R757	<i>forall-assignment-stmt</i>	<b>is</b> <i>assignment-stmt</i>
		<b>or</b> <i>pointer-assignment-stmt</i>
R758	<i>end-forall-stmt</i>	<b>is</b> END FORALL [ <i>forall-construct-name</i> ]
C737	(R758) If the <i>forall-construct-stmt</i> has a <i>forall-construct-name</i> , the <i>end-forall-stmt</i> shall have the same <i>forall-construct-name</i> . If the <i>end-forall-stmt</i> has a <i>forall-construct-name</i> , the <i>forall-construct-stmt</i> shall have the same <i>forall-construct-name</i> .	
C738	(R754) The <i>scalar-mask-expr</i> shall be scalar and of type logical.	
C739	(R754) Any procedure referenced in the <i>scalar-mask-expr</i> , including one referenced by a defined operation, shall be a pure procedure (12.7).	
C740	(R755) The <i>index-name</i> shall be a named scalar variable of type integer.	
C741	(R755) A <i>subscript</i> or <i>stride</i> in a <i>forall-triplet-spec</i> shall not contain a reference to any <i>index-name</i> in the <i>forall-triplet-spec-list</i> in which it appears.	
C742	(R756) A statement in a <i>forall-body-construct</i> shall not define an <i>index-name</i> of the <i>forall-construct</i> .	
C743	(R756) Any procedure referenced in a <i>forall-body-construct</i> , including one referenced by a defined	



operation, assignment, or finalization, shall be a pure procedure.

C744 (R756) A *forall-body-construct* shall not be a branch target.

C745 (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*

## Section 8:

R801 *block* **is** [ *execution-part-construct* ] ...

R802 *if-construct* **is** *if-then-stmt*  
                                   *block*  
                                   [ *else-if-stmt*  
                                   *block* ] ...  
                                   [ *else-stmt*  
                                   *block* ]  
                                   *end-if-stmt*

R803 *if-then-stmt* **is** [ *if-construct-name* : ] **IF** ( *scalar-logical-expr* ) **THEN**

R804 *else-if-stmt* **is** **ELSE IF** ( *scalar-logical-expr* ) **THEN** [ *if-construct-name* ]

R805 *else-stmt* **is** **ELSE** [ *if-construct-name* ]

R806 *end-if-stmt* **is** **END IF** [ *if-construct-name* ]

C801 (R802) 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*.

R807 *if-stmt* **is** **IF** ( *scalar-logical-expr* ) *action-stmt*

C802 (R807) The *action-stmt* in the *if-stmt* shall not be an *if-stmt*, *end-program-stmt*, *end-function-stmt*, or *end-subroutine-stmt*.

R808 *case-construct* **is** *select-case-stmt*  
                                   [ *case-stmt*  
                                   *block* ] ...  
                                   *end-select-stmt*

R809 *select-case-stmt* **is** [ *case-construct-name* : ] **SELECT CASE** ( *case-expr* )

R810 *case-stmt* **is** **CASE** *case-selector* [ *case-construct-name* ]

R811 *end-select-stmt* **is** **END SELECT** [ *case-construct-name* ]

C803 (R808) 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*.

R812 *case-expr* **is** *scalar-int-expr*  
                                   **or** *scalar-char-expr*  
                                   **or** *scalar-logical-expr*

R813 *case-selector* **is** ( *case-value-range-list* )  
                                   **or** **DEFAULT**

C804 (R808) No more than one of the selectors of one of the CASE statements shall be **DEFAULT**.

R814 *case-value-range* **is** *case-value*  
                                   **or** *case-value* :  
                                   **or** : *case-value*  
                                   **or** *case-value* : *case-value*

- R815 *case-value*                    **is** *scalar-int-initialization-expr*  
                                      **or** *scalar-char-initialization-expr*  
                                      **or** *scalar-logical-initialization-expr*
- C805 (R808) 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.
- C806 (R808) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.
- C807 (R808) 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*.
- R816 *associate-construct*           **is** *associate-stmt*  
                                              *block*  
                                              *end-associate-stmt*
- R817 *associate-stmt*                **is** [ *associate-construct-name* : ] ASSOCIATE ■  
                                              ■ ( *association-list* )
- R818 *association*                   **is** *associate-name* => *selector*
- R819 *selector*                       **is** *expr*  
                                      **or** *variable*
- C808 (R818) 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).
- C809 (R818) An *associate-name* shall not be the same as another *associate-name* in the same *associate-stmt*.
- R820 *end-associate-stmt*           **is** END ASSOCIATE [ *associate-construct-name* ]
- C810 (R820) 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*.
- R821 *select-type-construct*        **is** *select-type-stmt*  
                                              [ *type-guard-stmt*  
                                              *block* ] ...  
                                              *end-select-type-stmt*
- R822 *select-type-stmt*              **is** [ *select-construct-name* : ] SELECT TYPE ■  
                                              ■ ( [ *associate-name* => ] *selector* )
- C811 (R822) If *selector* is not a named *variable*, *associate-name* => shall appear.
- C812 (R822) 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).
- C813 (R822) The *selector* in a *select-type-stmt* shall be polymorphic.
- R823 *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* ]
- C814 (R823) The *type-spec* or *derived-type-spec* shall specify that each length type parameter is assumed.
- C815 (R823) The *type-spec* or *derived-type-spec* shall not specify a sequence derived type or a type with the BIND attribute.
- C816 (R823) If *selector* is not unlimited polymorphic, the *type-spec* or *derived-type-spec* shall specify an extension of the declared type of *selector*.
- C817 (R823) 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*.
- C818 (R823) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-guard-stmt*.
- R824 *end-select-type-stmt*                    **is**    END SELECT [ *select-construct-name* ]
- C819 (R821) 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*.
- R825 *do-construct*                            **is**    *block-do-construct*  
                                                  **or**    *nonblock-do-construct*
- R826 *block-do-construct*                    **is**    *do-stmt*  
                                                  *do-block*  
                                                  *end-do*
- R827 *do-stmt*                                **is**    *label-do-stmt*  
                                                  **or**    *nonlabel-do-stmt*
- R828 *label-do-stmt*                         **is**    [ *do-construct-name* : ] DO *label* [ *loop-control* ]
- R829 *nonlabel-do-stmt*                    **is**    [ *do-construct-name* : ] DO [ *loop-control* ]
- R830 *loop-control*                         **is**    [ , ] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■  
                                                  ■ [ , *scalar-int-expr* ]  
                                                  **or**    [ , ] WHILE ( *scalar-logical-expr* )  
                                                  **or**    [ , ] CONCURRENT *forall-header*
- R831 *do-variable*                           **is**    *scalar-int-variable-name*
- C820 (R831) The *do-variable* shall be a variable of type integer.
- R832 *do-block*                               **is**    *block*
- R833 *end-do*                                 **is**    *end-do-stmt*  
                                                  **or**    *continue-stmt*
- R834 *end-do-stmt*                         **is**    END DO [ *do-construct-name* ]
- C821 (R826) 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*.
- C822 (R826) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.
- C823 (R826) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.
- R835 *nonblock-do-construct*               **is**    *action-term-do-construct*  
                                                  **or**    *outer-shared-do-construct*
- R836 *action-term-do-construct*           **is**    *label-do-stmt*  
                                                  *do-body*  
                                                  *do-term-action-stmt*
- R837 *do-body*                               **is**    [ *execution-part-construct* ] ...
- R838 *do-term-action-stmt*                  **is**    *action-stmt*
- C824 (R838) A *do-term-action-stmt* shall not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.
- C825 (R835) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer to the same label.

- R839 *outer-shared-do-construct* **is** *label-do-stmt*  
*do-body*  
*shared-term-do-construct*
- R840 *shared-term-do-construct* **is** *outer-shared-do-construct*  
**or** *inner-shared-do-construct*
- R841 *inner-shared-do-construct* **is** *label-do-stmt*  
*do-body*  
*do-term-shared-stmt*
- R842 *do-term-shared-stmt* **is** *action-stmt*
- C826 (R842) A *do-term-shared-stmt* shall not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.
- C827 (R840) 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.
- R843 *cycle-stmt* **is** CYCLE [ *do-construct-name* ]
- C828 (R843) If a *cycle-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.
- C829 (R843) A *cycle-stmt* shall not appear within the range of a DO CONCURRENT construct if it belongs to an outer construct.
- R844 *exit-stmt* **is** EXIT [ *construct-name* ]
- C830 If an *exit-stmt* refers to a *construct-name*, it shall be within that construct; otherwise, it shall be within the range of at least one *do-construct*.
- C831 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.
- R845 *block-construct* **is** *block-stmt*  
[ *specification-part* ]  
*block*  
*end-block-stmt*
- R846 *block-stmt* **is** [ *block-construct-name* : ] BLOCK
- R847 *end-block-stmt* **is** END BLOCK [ *block-construct-name* ]
- C832 (R845) 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*.
- R848 *critical-construct* **is** *critical-stmt*  
*block*  
*end-critical-stmt*
- R849 *critical-stmt* **is** [ *critical-construct-name* : ] CRITICAL
- R850 *end-critical-stmt* **is** END CRITICAL [ *critical-construct-name* ]
- C833 (R848) 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*.
- C834 (R848) The *block* of a *critical-construct* shall not contain an image control statement.
- R851 *goto-stmt* **is** GO TO *label*
- C835 (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*
- C836 (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*
- C837 (R853) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.
- C838 (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*
- C839 (R856) The *scalar-char-initialization-expr* shall be of default kind.
- C840 (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*
- C841 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*
- C842 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** \*
- C843 An *image-set* that is an *int-expr* shall be scalar or of rank one and shall not be a co-indexed object.
- 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*
- C844 (R864) No specifier shall appear more than once in a given *query-spec-list*.
- R866 *sync-memory-stmt*            **is** SYNC\_MEMORY [ ( [ *sync-stat-list* ] ) ]

## Section 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 (R909) No specifier shall appear more than once in a given *close-spec-list*.
- C908 (R909) 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 *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 (R913) No specifier shall appear more than once in a given *io-control-spec-list*.
- C911 (R913) An *io-unit* shall be specified; 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 (R922) No specifier shall appear more than once in a given *wait-spec-list*.
- C940 (R922) 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 *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 (R926) No specifier shall appear more than once in a given *position-spec-list*.
- C943 (R926) 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 *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 (R928) No specifier shall appear more than once in a given *flush-spec-list*.
- C946 (R928) A *file-unit-number* shall be specified; 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** UNFORMATTED = *scalar-default-char-variable*  
**or** WRITE = *scalar-default-char-variable*

- C948 (R930) No specifier shall appear more than once in a given *inquire-spec-list*.
- C949 (R930) An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not both.
- C950 (R930) 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 (R930) If an ID= specifier appears, 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.

## Section 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 ] ...*

**Section 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*.  
 C1103 (R1101) An automatic object shall not appear in the *specification-part* (R204) of a main program.  
 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*  
 C1104 (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*.  
 C1105 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.  
 C1106 (R1104) An automatic object shall not appear in the *specification-part* of a module.  
 C1107 (R1104) If an object of a type for which *component-initialization* is specified (R448) appears in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER attribute, the object 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*  
 C1108 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.  
 C1109 (R1109) If *module-nature* is NON\_INTRINSIC, *module-name* shall be the name of a nonintrinsic module.  
 C1110 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the

same name.

C1111 (R1111) OPERATOR(*use-defined-operator*) shall not identify a *generic-binding*.

C1112 (R1112) The *generic-spec* shall not identify a *generic-binding*.

C1113 (R1112) Each *generic-spec* shall be a public entity in the module.

C1114 (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*

C1115 (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* ] ]

C1116 (R1116) An automatic object shall not appear in the *specification-part* of a submodule.

C1117 (R1116) A submodule *specification-part* shall not contain a *format-stmt* or a *stmt-function-stmt*.

C1118 (R1116) An object with default initialization that is declared in the *specification-part* of a submodule shall have the ALLOCATABLE, POINTER, or SAVE attribute.

C1119 (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.

C1120 (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* ] ]

C1121 (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*.

C1122 (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.

C1123 (R1120) A type declaration statement in a *block-data specification-part* shall not contain ALLOCATABLE, EXTERNAL, or BIND attribute specifiers.

## Section 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 ( = )  
                  **or** *dtio-generic-spec*
- R1208 *dtio-generic-spec*      **is** READ (FORMATTED)  
                  **or** READ (UNFORMATTED)  
                  **or** WRITE (FORMATTED)  
                  **or** WRITE (UNFORMATTED)
- R1209 *import-stmt*      **is** IMPORT [ [ :: ] *import-name-list*
- 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 (R1209) The IMPORT statement is allowed only in an *interface-body* that is not a module procedure interface body.
- C1211 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.
- C1212 (R1205) A module procedure interface body shall not appear in an abstract interface block.
- 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*

- R1213 *proc-attr-spec*      **or** *declaration-type-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 (R1223) In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a pure procedure (12.7).

C1234 (R1224) The *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

C1235 An actual argument that is a co-indexed object shall not be associated with a dummy argument that has the `ASYNCHRONOUS` attribute.

C1236 (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.

C1237 (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.

[illegible]

R1226    *function-stmt*                      is   [ *prefix* ] FUNCTION *function-name* ■  
                                                        ■ ( [ *dummy-arg-name-list* ] ) [ *suffix* ]

C1238 (R1226) 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.

C1239 (R1226) If RESULT appears, the *function-name* shall not appear in any specification statement in the scoping unit of the function subprogram.

R1227 *proc-language-binding-spec* is *language-binding-spec*

C1240 (R1227) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the *function-stmt* or *subroutine-stmt* of an interface body for an abstract interface or a dummy procedure.

C1241 (R1227) A *proc-language-binding-spec* shall not be specified for an internal procedure.

C1242 (R1227) 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.

R1228 *dummy-arg-name* **is** *name*

C1243 (R1228) A *dummy-arg-name* shall be the name of a dummy argument.

R1229 *prefix*                      **is** *prefix-spec* [ *prefix-spec* ] ...

R1230	<i>prefix-spec</i>	is	<i>declaration-type-spec</i>
		or	ELEMENTAL
		or	IMPURE



or MODULE  
or PURE  
or RECURSIVE

- |       |                                   |                                                                                                                                                                                                                                                                                               |
|-------|-----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C1244 | (R1229)                           | A <i>prefix</i> shall contain at most one of each <i>prefix-spec</i> .                                                                                                                                                                                                                        |
| C1245 | (R1229)                           | A <i>prefix</i> shall not specify both PURE and IMPURE.                                                                                                                                                                                                                                       |
| C1246 | (R1229)                           | A <i>prefix</i> shall not specify both ELEMENTAL and RECURSIVE.                                                                                                                                                                                                                               |
| C1247 | (R1229)                           | A <i>prefix</i> shall not specify ELEMENTAL if <i>proc-language-binding-spec</i> appears in the <i>function-stmt</i> or <i>subroutine-stmt</i> .                                                                                                                                              |
| C1248 | (R1229)                           | MODULE shall appear only within the <i>function-stmt</i> or <i>subroutine-stmt</i> of a module subprogram or of an interface body that is declared in the scoping unit of a module or submodule.                                                                                              |
| C1249 | (R1229)                           | If MODULE appears within the <i>prefix</i> 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. |
| C1250 | (R1229)                           | If MODULE appears within the <i>prefix</i> in a module subprogram, the subprogram shall specify the same characteristics and dummy argument names as its corresponding (12.6.2.4) module procedure interface body.                                                                            |
| C1251 | (R1229)                           | If MODULE appears within the <i>prefix</i> 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.                                                                                 |
| C1252 | (R1229)                           | If MODULE appears within the <i>prefix</i> in a module subprogram, RECURSIVE shall appear if and only if RECURSIVE appears in the <i>prefix</i> in the corresponding module procedure interface body.                                                                                         |
| R1231 | <i>suffix</i>                     | <b>is</b> <i>proc-language-binding-spec</i> [ RESULT ( <i>result-name</i> ) ]<br><b>or</b> RESULT ( <i>result-name</i> ) [ <i>proc-language-binding-spec</i> ]                                                                                                                                |
| R1232 | <i>end-function-stmt</i>          | <b>is</b> END [ FUNCTION [ <i>function-name</i> ] ]                                                                                                                                                                                                                                           |
| C1253 | (R1225)                           | An internal function subprogram shall not contain an ENTRY statement.                                                                                                                                                                                                                         |
| C1254 | (R1225)                           | An internal function subprogram shall not contain an <i>internal-subprogram-part</i> .                                                                                                                                                                                                        |
| C1255 | (R1232)                           | If a <i>function-name</i> appears in the <i>end-function-stmt</i> , it shall be identical to the <i>function-name</i> specified in the <i>function-stmt</i> .                                                                                                                                 |
| R1233 | <i>subroutine-subprogram</i>      | <b>is</b> <i>subroutine-stmt</i><br>[ <i>specification-part</i> ]<br>[ <i>execution-part</i> ]<br>[ <i>internal-subprogram-part</i> ]<br><i>end-subroutine-stmt</i>                                                                                                                           |
| R1234 | <i>subroutine-stmt</i>            | <b>is</b> [ <i>prefix</i> ] SUBROUTINE <i>subroutine-name</i> ■<br>■ [ ( [ <i>dummy-arg-list</i> ] ) [ <i>proc-language-binding-spec</i> ] ]                                                                                                                                                  |
| C1256 | (R1234)                           | The <i>prefix</i> of a <i>subroutine-stmt</i> shall not contain a <i>declaration-type-spec</i> .                                                                                                                                                                                              |
| R1235 | <i>dummy-arg</i>                  | <b>is</b> <i>dummy-arg-name</i><br><b>or</b> *                                                                                                                                                                                                                                                |
| R1236 | <i>end-subroutine-stmt</i>        | <b>is</b> END [ SUBROUTINE [ <i>subroutine-name</i> ] ]                                                                                                                                                                                                                                       |
| C1257 | (R1233)                           | An internal subroutine subprogram shall not contain an ENTRY statement.                                                                                                                                                                                                                       |
| C1258 | (R1233)                           | An internal subroutine subprogram shall not contain an <i>internal-subprogram-part</i> .                                                                                                                                                                                                      |
| C1259 | (R1236)                           | If a <i>subroutine-name</i> appears in the <i>end-subroutine-stmt</i> , it shall be identical to the <i>subroutine-name</i> specified in the <i>subroutine-stmt</i> .                                                                                                                         |
| R1237 | <i>separate-module-subprogram</i> | <b>is</b> MODULE PROCEDURE <i>procedure-name</i><br>[ <i>specification-part</i> ]<br>[ <i>execution-part</i> ]                                                                                                                                                                                |

[ *internal-subprogram-part* ]  
*end-sep-subprogram-stmt*

- R1238 *end-sep-subprogram-stmt*    **is**    **END** [PROCEDURE [*procedure-name*]]
- C1260 (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.
- C1261 (R1238) If a *procedure-name* appears in the *end-sep-subprogram-stmt*, it shall be identical to the *procedure-name* in the MODULE PROCEDURE statement.
- R1239 *entry-stmt*                            **is**    **ENTRY** *entry-name* [ ( [ *dummy-arg-list* ] ) [ *suffix* ] ]
- C1262 (R1239) If RESULT appears, the *entry-name* shall not appear in any specification or type-declaration statement in the scoping unit of the function program.
- C1263 (R1239) 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*.
- C1264 (R1239) RESULT shall appear only if the *entry-stmt* is in a function subprogram.
- C1265 (R1239) Within the subprogram containing the *entry-stmt*, the *entry-name* shall not appear as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY statement nor shall it appear in an EXTERNAL, INTRINSIC, or PROCEDURE statement.
- C1266 (R1239) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function subprogram.
- C1267 (R1239) 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.
- R1240 *return-stmt*                            **is**    **RETURN** [ *scalar-int-expr* ]
- C1268 (R1240) The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.
- C1269 (R1240) The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.
- R1241 *contains-stmt*                            **is**    **CONTAINS**
- R1242 *stmt-function-stmt*                    **is**    *function-name* ( [ *dummy-arg-name-list* ] ) = *scalar-expr*
- C1270 (R1242) 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, 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.
- C1271 (R1242) 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.
- C1272 (R1242) 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.
- C1273 (R1242) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.
- C1274 (R1242) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.
- C1275 (R1242) 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.
- C1276 The *specification-part* of a pure function subprogram shall specify that all its nonpointer dummy data objects have INTENT(IN).
- C1277 The *specification-part* of a pure subroutine subprogram shall specify the intents of all its non-pointer dummy data objects.
- C1278 A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure subpro-

gram shall not have the SAVE attribute.

- C1279 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are pure.
- C1280 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.
- C1281 All internal subprograms in a pure subprogram shall be pure.
- C1282 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
- C1283 Any procedure referenced in a pure subprogram, including one referenced via a defined operation, assignment, or finalization, shall be pure.
- C1284 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*.
- C1285 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-number* or \*.
- C1286 A pure subprogram shall not contain a *stop-stmt*.
- C1287 A co-indexed object shall not appear in a variable definition context in a pure subprogram.
- C1288 A pure subprogram shall not contain an image control statement (8.5.1).
- C1289 All dummy arguments of an elemental procedure shall be scalar dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.
- C1290 The result variable of an elemental function shall be scalar and shall not have the POINTER or ALLOCATABLE attribute.
- C1291 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 argument to one of the intrinsic functions BIT\_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

### Section 13:

### Section 14:

### Section 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.

### Section 16:

## D.2 Syntax rule cross-reference

R475	<i>ac-do-variable</i>	R474, C508, R474, C508
R473	<i>ac-implied-do</i>	R472, C508, R472, C508
R474	<i>ac-implied-do-control</i>	R473
R469	<i>ac-spec</i>	R468
R472	<i>ac-value</i>	R469, R473, C505, C506, C507, R469, R473, C505, C506, C507

R521	<i>access-id</i>	R520, C558, R520, C558
R507	<i>access-spec</i>	R316, R432, R442, R447, R455, R456, R432, R442, R447, R455, R456, R502, C516, R520, R502, C516, R520, R1213 R212, C558
R520	<i>access-stmt</i>	R213, C324, C325, C324, C325, R807, C802, R807, C802
R214	<i>action-stmt</i>	R835
R836	<i>action-term-do-construct</i>	R1222, C1233, R1222, C1233
R1223	<i>actual-arg</i>	C501, R1219, C1223, R1220, C1229, R1219, C1223, R1220, C1229
R1222	<i>actual-arg-spec</i>	R310, R706
R709	<i>add-op</i>	R705, R706, R705, R706
R705	<i>add-operand</i>	R626, C634, R626, C634
R627	<i>alloc-opt</i>	R522
R523	<i>allocatable-decl</i>	R212
R522	<i>allocatable-stmt</i>	R631, C632, C633, R631, C632, C633
R636	<i>allocate-co-array-spec</i>	R636, C633, R636, C633
R637	<i>allocate-co-shape-spec</i>	R631, C626, C627, C628, C629, C630, C631, C632, C633, C636, C637, C638, R640, C640, R631, C626, C627, C628, C629, C630, C631, C632, C633, C636, C637, C638, R640, C640
R632	<i>allocate-object</i>	R631, C632, C633, R631, C632, C633
R633	<i>allocate-shape-spec</i>	R214
R626	<i>allocate-stmt</i>	R626
R631	<i>allocation</i>	R301, R304, R301, R304
R302	<i>alphanumeric-character</i>	C1223, R1223, C1223, R1223
R1224	<i>alt-return-spec</i>	R1118, C1119, R1118, C1120
—	<i>ancestor-module-name</i>	R310, R715
R719	<i>and-op</i>	R715
R714	<i>and-operand</i>	R447, C458, R456, C479, R447, C458, R456, C479
—	<i>arg-name</i>	R343
R344	<i>arg-token</i>	R214, C325, C824, C826, C837, C824, C826, C837
R853	<i>arithmetic-if-stmt</i>	C505, C506, C507, C505, C506, C507, R701
R468	<i>array-constructor</i>	R532, C565, C568, R562, R532, C565, C568, R562, R603, R610, R603, R610
R617	<i>array-element</i>	R539, R540, R539, R540
—	<i>array-name</i>	R603, C622, R603, C622
R618	<i>array-section</i>	R502, R503, C514, R509, C529, R522, R523, R539, R540, R551, R552, R564, C593, R502, R503, C514, R509, C529, R522, R523, R539, R540, R551, R552, R564, C593
R510	<i>array-spec</i>	R214, R747, R757, C745, R747, R757, C744
R734	<i>assignment-stmt</i>	R213, C810
R816	<i>associate-construct</i>	R817, R820, C810, R817, R820, C810
—	<i>associate-construct-name</i>	R818, C808, C809, R822, C811, C812, R818, C808, C809, R822, C811, C812
—	<i>associate-name</i>	R816, C809, C810, R816, C809, C810
R817	<i>associate-stmt</i>	R817
R818	<i>association</i>	R510
R515	<i>assumed-shape-spec</i>	R510, R511, R510, R511
R517	<i>assumed-size-spec</i>	R212
R524	<i>asynchronous-stmt</i>	

R502	<i>attr-spec</i>	R501, C501, C575, R501, C501, C575
R923	<i>backspace-stmt</i>	R214, C1284
R342	<i>basic-token</i>	R341, R344, R341, R344
R341	<i>basic-token-sequence</i>	R337, R341, R337, R341
R426	<i>binary-constant</i>	R425
R526	<i>bind-entity</i>	R525, C560, R525, C560
R525	<i>bind-stmt</i>	R212
R456	<i>binding-attr</i>	R454, C477, C480, C481, R454, C477, C480, C481
—	<i>binding-name</i>	R454, R455, C472, R454, R455, C472, R1221, C1226, R1221, C1226
R452	<i>binding-private-stmt</i>	R451, C467, R451, C467
R1017	<i>blank-interp-edit-desc</i>	R1012
R801	<i>block</i>	R802, R808, R816, R821, R832, R845, R848, C834, R802, R808, R816, R821, R832, R845, R848, C834
R845	<i>block-construct</i>	R213, C832
—	<i>block-construct-name</i>	R846, R847, C832, R846, R847, C832
R1120	<i>block-data</i>	R202, C1122, C1123, C1124
—	<i>block-data-name</i>	R1121, R1122, C1121, R1121, R1122, C1122
R1121	<i>block-data-stmt</i>	R1120, C1121, R1120, C1122
R826	<i>block-do-construct</i>	R825, C821, R825, C821
R846	<i>block-stmt</i>	R845, C832, R845, C832
R738	<i>bounds-remapping</i>	R735, C720, C721, R735, C720, C721
R737	<i>bounds-spec</i>	R735, C719, R735, C719
R425	<i>boz-literal-constant</i>	R306, C414
R1220	<i>call-stmt</i>	R214, C1234
R808	<i>case-construct</i>	R213, C803, C805, C807, C803, C805, C807
—	<i>case-construct-name</i>	R809, R810, R811, C803, R809, R810, R811, C803
R812	<i>case-expr</i>	R809, C805, C806, C807, R809, C805, C806, C807
R813	<i>case-selector</i>	R810
R810	<i>case-stmt</i>	R808, C803, R808, C803
R815	<i>case-value</i>	R814, C805, R814, C805
R814	<i>case-value-range</i>	R813, C806, C807, R813, C806, C807
R309	<i>char-constant</i>	C303
R725	<i>char-expr</i>	C706, R731, C706, R731, R812
R731	<i>char-initialization-expr</i>	R508, C521, R508, C521, C713, R815, R856, C839, R815, R856, C839, R913, C925, R913, C925
R422	<i>char-length</i>	R421, C420, C453, R421, C420, C453, R503, C504, R503, C504
R423	<i>char-literal-constant</i>	R306, R1006, C1009, R1020, C1013, 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, R903, C901, C902
R909	<i>close-spec</i>	R908, C907, C908, R908, C907, C908
R908	<i>close-stmt</i>	R214, C1284
—	<i>co-array-name</i>	R540
R511	<i>co-array-spec</i>	R444, C447, R444, C447, R503, R509, C523, R523, R540, R552, R503, R509, C523, R523, R540, R552

R625	<i>co-subscript</i>	R624
—	<i>co-subscript-list</i>	R624
—	<i>common-block-name</i>	R526, R549, R563, R526, R549, R563
R564	<i>common-block-object</i>	R563, C593, C594, C595, C596, R563, C593, C594, C595, C596
R563	<i>common-stmt</i>	R212
R417	<i>complex-literal-constant</i>	R306
R615	<i>complex-part-designator</i>	R603, R618, C622, R603, R618, C622
R445	<i>component-array-spec</i>	R442, R444, C446, C449, R442, R444, C446, C449
R442	<i>component-attr-spec</i>	R441, C443, C463, R441, C443, C463
R462	<i>component-data-source</i>	R461
R443	<i>component-decl</i>	R441, C461, R441, C461
R440	<i>component-def-stmt</i>	R439, C443, C444, C445, C454, R439, C443, C444, C445, C454
R444	<i>component-dim-spec</i>	R442
R448	<i>component-initialization</i>	C461, C462, C463, C461, C462, C463, C1107
—	<i>component-name</i>	C464
R439	<i>component-part</i>	R430
R461	<i>component-spec</i>	R460, C495, C496, C497, C498, C500, C501, R460, C495, C496, C497, C498, C500, C501
R852	<i>computed-goto-stmt</i>	R214, C836
R711	<i>concat-op</i>	R310, R710
R905	<i>connect-spec</i>	R904, C903, C904, R904, C903, C904
R305	<i>constant</i>	R308, R309, R308, R309, R536, R610, R701
R538	<i>constant-subobject</i>	R536, R537, C573, R536, R537, C573
—	<i>construct-name</i>	R844, C830, R844, C830
R1241	<i>contains-stmt</i>	R210, R451, R1107
R854	<i>continue-stmt</i>	R214, C325, R833, C824, R833, C824
R1012	<i>control-edit-desc</i>	R1003
R848	<i>critical-construct</i>	R213, C833, C834, C833, C834
—	<i>critical-construct-name</i>	R849, R850, C833, R849, R850, C833
R849	<i>critical-stmt</i>	R848, C833, R848, C833
R843	<i>cycle-stmt</i>	R214, C325, C824, C826, C828, C829, C824, C826, C828, C829
R1009	<i>d</i>	R1006, C1007, C1008, R1006, C1007, C1008
R441	<i>data-component-def-stmt</i>	R440
R1006	<i>data-edit-desc</i>	R1003
R532	<i>data-i-do-object</i>	R531, C561, C563, C564, C568, R531, C561, C563, C564, C568
R533	<i>data-i-do-variable</i>	R531, C568, R531, C568
R531	<i>data-implied-do</i>	R530, R532, C568, R530, R532, C568
—	<i>data-pointer-component-name</i>	R736, C724, R736, C724
R736	<i>data-pointer-object</i>	R735, C717, C718, C719, C720, C721, C725, R735, C717, C718, C719, C720, C721, C725
R612	<i>data-ref</i>	C612, C616, R614, R617, R618, C612, C616, R614, R617, R618, C723, C729, C723, R1221, C1226, C1227, R1221, C1226, C1227
R528	<i>data-stmt</i>	R209, R212, R209, R212, C1206
R536	<i>data-stmt-constant</i>	C414, R534, C571, R534, C571

R530	<i>data-stmt-object</i>	R529, C561, C562, C563, C564, R529, C561, C562, C563, C564
R535	<i>data-stmt-repeat</i>	R534, C569, R534, C569
R529	<i>data-stmt-set</i>	R528
R534	<i>data-stmt-value</i>	R529
R739	<i>data-target</i>	R462, C502, C503, R462, C502, C503, C546, R735, C717, C718, C721, C727, R735, C717, C718, C721, C727
R641	<i>dealloc-opt</i>	R640, C641, R640, C641
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, C424, R441, C444, C445, C404, C405, C424, R441, C444, C445, R501, R556, R501, R556, R1212, R1230, C1256, R1212, R1230, C1256
R726	<i>default-char-expr</i>	C707, R905, R906, R909, R913, R914, R905, R906, R909, R913, R914
R607	<i>default-char-variable</i>	C607, R629, C607, R629, R907, R930, R907, R930
R605	<i>default-logical-variable</i>	C605, R930
R516	<i>deferred-shape-spec</i>	R444, R445, C446, C447, R444, R445, C446, C447, C523, R510, R511, C529, R546, R510, R511, C529, R546
—	<i>deferred-shape-spec-list</i>	C523
R315	<i>define-macro-stmt</i>	R314
R723	<i>defined-binary-op</i>	R311, R722, C704, R722, C704, R1114, R1115, R1114, R1115
R311	<i>defined-operator</i>	C474, R1207, C1202, R1207, C1202
R703	<i>defined-unary-op</i>	R311, R702, C703, R702, C703, R1114, R1115, R1114, R1115
R430	<i>derived-type-def</i>	R207, C437, C441, C442, C437, C441, C442
R458	<i>derived-type-spec</i>	R402, C403, R403, C405, C406, R460, C494, C501, R402, C403, R403, C405, C406, R460, C494, C501, R823, C814, C815, C816, R823, C814, C815, C816, R920, C937, C938, R920, C937, C938
R431	<i>derived-type-stmt</i>	R430, C432, C438, C441, C442, R430, C432, C438, C441, C442
R603	<i>designator</i>	R449, C465, R449, C465, R538, R601, C601, R615, C619, R616, C620, R601, C601, R615, C619, R616, C620, R701, C702, C745, R701, C702, C744
—	<i>digit</i>	R302, R313, R302, R313, R410, R426, C412, R427, C413, R429, R426, C429, R427, C430, R429, R410, R426, C412, R427, C413, R429, R426, C429, R427, C430, R429
R410	<i>digit-string</i>	R407, R408, R409, R413, R414, R407, R408, R409, R413, R414
R540	<i>dimension-decl</i>	R539
R539	<i>dimension-stmt</i>	R212
R832	<i>do-block</i>	R826
R837	<i>do-body</i>	R836, R839, R841, R836, R839, R841
R825	<i>do-construct</i>	R213, C828, C830, C828, C830
—	<i>do-construct-name</i>	R828, R829, R834, C821, R843, C828, R828, R829, R834, C821, R843, C828
R827	<i>do-stmt</i>	R826, C821, C822, C823, R826, C821, C822, C823
R838	<i>do-term-action-stmt</i>	C325, R836, C824, C825, R836, C824, C825

R842	<i>do-term-shared-stmt</i>	R841, C826, C827, R841, C826, C827
R831	<i>do-variable</i>	R475, R533, R830, C820, R830, C820, R919, C934, R919, C934
R1208	<i>dtio-generic-spec</i>	C476, R1207, C1202, R1207, C1202
R1235	<i>dummy-arg</i>	R1234, R1239, C1266, R1234, R1239, C1266
R1228	<i>dummy-arg-name</i>	R541, R542, R553, R541, R542, R553, R1226, C1243, R1235, R1242, C1272, C1273, C1274, R1226, C1243, R1235, R1242, C1272, C1273, C1274
R1010	<i>e</i>	R1006, C1005, C1008, R1006, C1005, C1008
R804	<i>else-if-stmt</i>	R802, C801, R802, C801
R805	<i>else-stmt</i>	R802, C801, R802, C801
R750	<i>elsewhere-stmt</i>	R744, C735, R744, C734
R820	<i>end-associate-stmt</i>	R816, C810, R816, C810
R1122	<i>end-block-data-stmt</i>	R1120, C1121, R1120, C1122
R847	<i>end-block-stmt</i>	R845, C832, R845, C832
R850	<i>end-critical-stmt</i>	R848, C833, R848, C833
R833	<i>end-do</i>	R826, C821, C822, C823, R826, C821, C822, C823
R834	<i>end-do-stmt</i>	R833, C821, C822, R833, C821, C822
R467	<i>end-enum-stmt</i>	R463
R758	<i>end-forall-stmt</i>	R752, C737, R752, C736
R1232	<i>end-function-stmt</i>	R214, C201, R214, C201, C324, C325, C324, C325, C802, C824, C826, C802, C824, C826, R1205, R1225, C1255, R1205, R1225, C1255
R806	<i>end-if-stmt</i>	R802, C801, R802, C801
R1204	<i>end-interface-stmt</i>	R1201, C1202, R1201, C1202
R336	<i>end-macro-stmt</i>	R314
R1106	<i>end-module-stmt</i>	R1104, C1104, R1104, C1104
R1103	<i>end-program-stmt</i>	R214, C201, R214, C201, C324, C325, C324, C325, C802, C824, C826, C802, C824, C826, R1101, C1102, R1101, C1102
R811	<i>end-select-stmt</i>	R808, C803, R808, C803
R824	<i>end-select-type-stmt</i>	R821, C819, R821, C819
R1238	<i>end-sep-subprogram-stmt</i>	R1237, C1261, R1237, C1261
R1119	<i>end-submodule-stmt</i>	R1116, C1120, R1116, C1121
R1236	<i>end-subroutine-stmt</i>	R214, C201, R214, C201, C324, C325, C324, C325, C802, C824, C826, C802, C824, C826, R1205, R1233, C1259, R1205, R1233, C1259
R434	<i>end-type-stmt</i>	R430
R751	<i>end-where-stmt</i>	R744, C735, R744, C734
R924	<i>endfile-stmt</i>	R214, C1284
R503	<i>entity-decl</i>	R501, C502, C503, C505, C551, R501, C502, C503, C505
—	<i>entity-decl-list</i>	C551
—	<i>entity-name</i>	R526, R547, R526, R547
—	<i>entry-name</i>	C1238, R1239, C1262, C1265, C1267, C1238, R1239, C1262, C1265, C1267
R1239	<i>entry-stmt</i>	R206, R207, R209, R206, R207, R209, C1105, C1206, C1263, C1264, C1265, C1206, C1263, C1264, C1265
R463	<i>enum-def</i>	R207
R464	<i>enum-def-stmt</i>	R463



R466	<i>enumerator</i>	R465, C504, R465, C504
R465	<i>enumerator-def-stmt</i>	R463
R721	<i>equiv-op</i>	R310, R717
R716	<i>equiv-operand</i>	R716, R717, R716, R717
R562	<i>equivalence-object</i>	R561, C581, C582, C583, C584, C585, C586, C587, C588, C589, C590, C591, R561, C581, C582, C583, C584, C585, C586, C587, C588, C589, C590, C591
R561	<i>equivalence-set</i>	R560
R560	<i>equivalence-stmt</i>	R212
R629	<i>errmsg-variable</i>	R627, R641, R627, R641, R858
R213	<i>executable-construct</i>	R208, R209, R208, R209, C1263
R208	<i>execution-part</i>	C201, R1101, C1101, R1101, C1101, R1225, R1233, R1237, R1225, R1233, R1237
R209	<i>execution-part-construct</i>	R208, R801, R837, R801, R837
R844	<i>exit-stmt</i>	R214, C325, C824, C826, C830, C831, C824, C826, C830, C831
R338	<i>expand-stmt</i>	R322, C330, R322, C330
R512	<i>explicit-shape-spec</i>	R445, C449, C450, R445, C449, C450, C514, R510, C528, R517, C593, C514, R510, C528, R517, C593
R416	<i>exponent</i>	R413
R415	<i>exponent-letter</i>	R413, C415, R413, C415
R722	<i>expr</i>	R462, R472, R462, R472, R601, C602, R630, R601, C602, R630, R701, R722, R724, R725, R726, R727, R728, R730, R734, R739, C728, R742, C731, R701, R722, R724, R725, R726, R727, R728, R730, R734, R739, C728, R742, C730, R819, R916, R1223, R1242, C1270, C1271, C1275, R1223, R1242, C1270, C1271, C1275
R312	<i>extended-intrinsic-op</i>	R311
—	<i>external-name</i>	R1210
R1210	<i>external-stmt</i>	R212
R203	<i>external-subprogram</i>	R202, C1263
R906	<i>file-name-expr</i>	R905, R930, 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, R901, R905, C904, C906, R909, C908, C920, C926, R922, C940, R923, R924, R925, R926, C943, R927, R928, C946, R930, C950, C1285
R457	<i>final-binding</i>	R453
—	<i>final-subroutine-name</i>	R457, C485, C486, R457, C485, C486
R928	<i>flush-spec</i>	R927, C945, C946, R927, C945, C946
R927	<i>flush-stmt</i>	R214, C1284
R757	<i>forall-assignment-stmt</i>	R756, C745, R759, R756, C744, R759
R756	<i>forall-body-construct</i>	R752, C742, C743, C744, R752, C741, C742, C743
R752	<i>forall-construct</i>	R213, R756, C742, R756, C741
—	<i>forall-construct-name</i>	R753, R758, C737, R753, R758, C736
R753	<i>forall-construct-stmt</i>	R752, C737, R752, C736
R754	<i>forall-header</i>	R753, R759, R753, R759, R830
R759	<i>forall-stmt</i>	R214, R756
R755	<i>forall-triplet-spec</i>	R754, C741, R754, C740

R914	<i>format</i>	R910, R912, R913, C917, C918, C921, C929, C930, R910, R912, R913, C917, C918, C921, C929, C930
R1003	<i>format-item</i>	R1002, C1002, R1003, R1004, R1002, C1002, R1003, R1004
R1002	<i>format-specification</i>	R1001
R1001	<i>format-stmt</i>	R206, R207, R209, R206, R207, R209, C1001, C1105, C1117, C1105, C1118, C1206
—	<i>function-name</i>	R503, C508, R503, C508, C1203, R1226, C1238, C1239, R1232, C1255, C1267, R1242, C1273, C1203, R1226, C1238, C1239, R1232, C1255, C1267, R1242, C1273
R1219	<i>function-reference</i>	R506, C512, R506, C512, R701
R1226	<i>function-stmt</i>	R1205, C1203, R1225, C1240, C1247, C1248, C1255, R1205, C1203, R1225, C1240, C1247, C1248, C1255
R1225	<i>function-subprogram</i>	R203, R211, R203, R211, R1108
R455	<i>generic-binding</i>	R453, C471, R453, C471, C1111, C1112, C1113
—	<i>generic-name</i>	C473, R1207, C1202, R1207, C1202
R1207	<i>generic-spec</i>	R455, C471, C473, C474, C475, C476, R455, C471, C473, C474, C475, C476, R521, R1112, C1112, C1113, R1112, C1113, C1114, R1203, R1204, C1202, C1204, R1203, R1204, C1202, C1204
R851	<i>goto-stmt</i>	R214, C325, C824, C826, C835, C824, C826, C835
R428	<i>hex-constant</i>	R425
R429	<i>hex-digit</i>	R428, R1021
R802	<i>if-construct</i>	R213, C801
—	<i>if-construct-name</i>	R803, R804, R805, R806, C801, R803, R804, R805, R806, C801
R807	<i>if-stmt</i>	R214, C324, C802
R803	<i>if-then-stmt</i>	R802, C801, R802, C801
R419	<i>imag-part</i>	R417
R624	<i>image-selector</i>	R613, C615, R613, C615
R862	<i>image-set</i>	C843, R863, R864, C843, R863, R864
R860	<i>image-team</i>	C842, R905
R205	<i>implicit-part</i>	R204
R206	<i>implicit-part-stmt</i>	R205
R556	<i>implicit-spec</i>	R555
R555	<i>implicit-stmt</i>	R205, R206, R205, R206
R518	<i>implied-shape-spec</i>	R510
—	<i>import-name</i>	R1209, C1211, R1209, C1211
R1209	<i>import-stmt</i>	R204
—	<i>index-name</i>	R755, C740, C741, C742, R755, C739, C740, C741
R449	<i>initial-data-target</i>	R448, C464, R448, C464, R505, C511, R536, R505, C511, R536
R505	<i>initialization</i>	R503, C505, C506, C507, C510, R503, C505, C506, C507, C510
R730	<i>initialization-expr</i>	R448, R505, R544, R505, R544, C712
R841	<i>inner-shared-do-construct</i>	R840, C827, R840, C827
R915	<i>input-item</i>	R910, C916, R918, C932, C933, C935, R910, C916, R918, C932, C933, C935
R930	<i>inquire-spec</i>	R929, C948, C949, C950, R929, C948, C949, C950
R929	<i>inquire-stmt</i>	R214, C1284

R308	<i>int-constant</i>	C302, R535
—	<i>int-constant-name</i>	R408, C409, R408, C409
R537	<i>int-constant-subobject</i>	R535, C572, R535, C572
R727	<i>int-expr</i>	R401, R474, R401, R474, R611, R619, R622, R623, R625, R634, R635, R611, R619, R622, R623, R625, R634, R635, C708, R729, C711, R732, C708, R729, C711, R732, R812, R830, R852, R862, C843, R812, R830, R852, R862, C843, R902, R905, R913, R919, R922, R930, R902, R905, R913, R919, R922, R930, R1240, C1269, R1240, C1269
R732	<i>int-initialization-expr</i>	R405, C408, R420, C418, R437, R466, R405, C408, R420, C418, R437, R466, R531, C714, R815, R856, C840, R815, R856, C840
R407	<i>int-literal-constant</i>	R306, R406, R422, C419, R406, R422, C419, R1005, R1007, R1008, R1009, R1010, R1015, R1005, R1007, R1008, R1009, R1010, R1015
R608	<i>int-variable</i>	C608, R628, C608, R628, R905, R909, R913, R922, R926, R928, R929, R930, R905, R909, R913, R922, R926, R928, R929, R930
—	<i>int-variable-name</i>	R831
R519	<i>intent-spec</i>	R502, R541, R502, R541, R1213
R541	<i>intent-stmt</i>	R212
R1201	<i>interface-block</i>	R207, C1201
R1205	<i>interface-body</i>	R1202, C1201, C1205, C1206, C1210, R1202, C1201, C1205, C1206, C1210
R1215	<i>interface-name</i>	R454, C469, C482, R454, C469, C482, R1212, C1220, R1212, C1220
R1202	<i>interface-specification</i>	R1201
R1203	<i>interface-stmt</i>	R1201, C1202, C1203, R1201, C1202, C1203
R903	<i>internal-file-variable</i>	R901, C922, R901, C922
R211	<i>internal-subprogram</i>	R210
R210	<i>internal-subprogram-part</i>	R1101, R1225, C1254, R1233, C1258, R1237, C1278, R1225, C1254, R1233, C1258, R1237, C1278
R310	<i>intrinsic-operator</i>	R312, C703, C704, C703, C704
—	<i>intrinsic-procedure-name</i>	R1218, C1221, R1218, C1221
R1218	<i>intrinsic-stmt</i>	R212
R404	<i>intrinsic-type-spec</i>	R402, R403, R402, R403
R913	<i>io-control-spec</i>	R910, R911, C910, C911, C917, C918, C919, C920, C928, R910, R911, C910, C911, C917, C918, C919, C920, C928
R917	<i>io-implied-do</i>	R915, R916, R915, R916
R919	<i>io-implied-do-control</i>	R917
R918	<i>io-implied-do-object</i>	R917, C935, R917, C935
R901	<i>io-unit</i>	R913, C911, C918, C919, C920, C922, C926, R913, C911, C918, C919, C920, C922, C926, C1285
R907	<i>iomsg-variable</i>	R905, R909, R913, R922, R926, R928, R930, R905, R909, R913, R922, R926, R928, R930
R1013	<i>k</i>	R1012, C1010, R1012, C1010
R215	<i>keyword</i>	R459, C491, C492, R461, C498, C499, R459, C491, C492, R461, C498, C499, R1222, C1228, C1229, C1230, R1222, C1228, C1229, C1230

R408	<i>kind-param</i>	R407, C410, C411, R413, C415, C416, C419, R423, C427, R424, C428, R425, R407, C410, C411, R413, C415, C416, C419, R423, C427, R424, C428, R425
R405	<i>kind-selector</i>	R404, R436, R404, R436
R313	<i>label</i>	C304, R828, C823, R851, C835, R852, C836, R853, C837, R828, C823, R851, C835, R852, C836, R853, C837, R905, C905, R909, C909, R913, C914, R914, C931, R922, C941, R926, C944, R928, C947, R930, C952, R905, C905, R909, C909, R913, C914, R914, C931, R922, C941, R926, C944, R928, C947, R930, C952, R1224, C1234, R1224, C1234
R828	<i>label-do-stmt</i>	R827, C823, R836, C825, R839, R841, C827, R827, C823, R836, C825, R839, R841, C827
R508	<i>language-binding-spec</i>	R502, C502, C503, R525, C560, R502, C502, C503, R525, C560, R1227
R470	<i>lbracket</i>	R343, R444, R468, R444, R468, R503, R509, R523, R540, R552, R503, R509, R523, R540, R552, R624, R631, R624, R631
R421	<i>length-selector</i>	R420, C424, C425, R420, C424, C425
—	<i>letter</i>	R302, R304, R302, R304, R557, C577, R557, C577, R703, R723, R703, R723
R557	<i>letter-spec</i>	R556
R702	<i>level-1-expr</i>	R704
R706	<i>level-2-expr</i>	R706, R710, R706, R710
R710	<i>level-3-expr</i>	R710, R712, R710, R712
R712	<i>level-4-expr</i>	R714
R717	<i>level-5-expr</i>	R717, R722, 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, C705, R733, R748, R803, R804, R807, R812, R830, R803, R804, R807, R812, R830
R733	<i>logical-initialization-expr</i>	C715, R815
R424	<i>logical-literal-constant</i>	R306, C703, C704, C703, C704
R604	<i>logical-variable</i>	C604
R830	<i>loop-control</i>	R828, R829, R828, R829
R513	<i>lower-bound</i>	R512, R515, R517, R518, R512, R515, R517, R518
R634	<i>lower-bound-expr</i>	R633, R636, R637, R633, R636, R637, R737, R738, R737, R738
R1008	<i>m</i>	R1006, C1008, R1006, C1008
R339	<i>macro-actual-arg</i>	C327, C328, C330, C327, C328, C330
R321	<i>macro-body-block</i>	R314, R323, R327, R314, R323, R327
R322	<i>macro-body-construct</i>	R321, C309, R321, C309
R333	<i>macro-body-stmt</i>	R322, C315, C316, R322, C315, C316
R332	<i>macro-condition</i>	R328, R329, R328, R329
R317	<i>macro-declaration-stmt</i>	R314
R314	<i>macro-definition</i>	R207, R322, C309, R322, C309
R323	<i>macro-do-construct</i>	R322
R325	<i>macro-do-limit</i>	C311
R324	<i>macro-do-stmt</i>	R323
—	<i>macro-do-variable-name</i>	C310

——	<i>macro-dummy-arg-name</i>	C305, C307, C305, C307
——	<i>macro-dummy-arg-name-list</i>	C305
——	<i>macro-dummy-name</i>	C329, C330, C329, C330
R329	<i>macro-else-if-stmt</i>	R327
R330	<i>macro-else-stmt</i>	R327
R326	<i>macro-end-do-stmt</i>	R323
R331	<i>macro-end-if-stmt</i>	R327
R337	<i>macro-expr</i>	R320, C308, R325, R332, C318, R320, C308, R325, R332, C318
R327	<i>macro-if-construct</i>	R322
R328	<i>macro-if-then-stmt</i>	R327
——	<i>macro-local-variable-name</i>	C306
——	<i>macro-name</i>	R336, C317, C319, R336, C317, C319
R318	<i>macro-type-declaration-stmt</i>	R317
R1101	<i>main-program</i>	R202, C1101
R748	<i>mask-expr</i>	R743, R745, R749, R754, C738, C739, R743, R745, R749, R754, C737, C738
R749	<i>masked-elsewhere-stmt</i>	R744, C735, R744, C734
R1104	<i>module</i>	R202
——	<i>module-name</i>	R1105, R1106, C1104, R1109, C1108, C1109, R1105, R1106, C1104, R1109, C1108, C1109, C1111
R1110	<i>module-nature</i>	R1109, C1108, C1109, R1109, C1108, C1109
R1105	<i>module-stmt</i>	R1104, C1104, R1104, C1104
R1108	<i>module-subprogram</i>	R1107, C1263
R1107	<i>module-subprogram-part</i>	R1104, R1116, R1104, R1116
R708	<i>mult-op</i>	R310, R705
R704	<i>mult-operand</i>	R704, R705, R704, R705
R1015	<i>n</i>	R1014, C1011, C1012, R1014, C1011, C1012
R304	<i>name</i>	R102, R215, C301, R307, C301, R307, R504, R550, R504, R550, R602, R1215, C1213, C1214, R1228, R1215, C1213, C1214, R1228
R307	<i>named-constant</i>	R305, R418, R419, R466, R418, R419, R466, R544
R544	<i>named-constant-def</i>	R543
——	<i>namelist-group-name</i>	R558, C578, C580, R558, C578, C580, R913, C915, C916, C917, C919, C921, C929, C930, R913, C915, C916, C917, C919, C921, C929, C930
R559	<i>namelist-group-object</i>	R558, C579, C580, R558, C579, C580
R558	<i>namelist-stmt</i>	R212
R343	<i>nested-token-sequence</i>	R341
R835	<i>nonblock-do-construct</i>	R825
R829	<i>nonlabel-do-stmt</i>	R827, C822, R827, C822
R718	<i>not-op</i>	R310, R714
R863	<i>notify-stmt</i>	R214
R506	<i>null-init</i>	R448, R505, R536, R505, R536, R1216
R638	<i>nullify-stmt</i>	R214
R728	<i>numeric-expr</i>	C709, R853, C838, R853, C838
R504	<i>object-name</i>	R503, C506, C509, C511, R522, R523, R524, R527, R546, R549, R551, R552, R554, R503, C506, C509, C511, R522, R523, R524, R527, R546, R549, R551, R552, R554, 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, C1284
R542	<i>optional-stmt</i>	R212
R720	<i>or-op</i>	R310, R716
R715	<i>or-operand</i>	R715, R716, R715, R716
R839	<i>outer-shared-do-construct</i>	R835, R840, C827, R835, R840, C827
R916	<i>output-item</i>	R911, R912, C916, R918, C935, C936, R929, R911, R912, C916, R918, C935, C936, R929
R543	<i>parameter-stmt</i>	R206, R207, R206, R207
R1118	<i>parent-identifier</i>	R1117
R610	<i>parent-string</i>	R609, C609, R609, C609
—	<i>parent-submodule-name</i>	R1118, C1119, R1118, C1120
—	<i>parent-type-name</i>	R432, C433, R432, C433
—	<i>part-name</i>	R613, C610, C611, C612, C613, C614, C615, C617, C618, C623, R613, C610, C611, C612, C613, C614, C615, C617, C618, C623
R613	<i>part-ref</i>	C564, C567, C582, C564, C567, C582, R612, C617, C618, C621, C622, R612, C617, C618, C621, C622
R735	<i>pointer-assignment-stmt</i>	R214, C546, R757
R546	<i>pointer-decl</i>	R545
R639	<i>pointer-object</i>	R638, C639, R638, C639
R545	<i>pointer-stmt</i>	R212
R1014	<i>position-edit-desc</i>	R1012
R926	<i>position-spec</i>	R923, R924, R925, C942, C943, R923, R924, R925, C942, C943
R707	<i>power-op</i>	R310, R704
R1229	<i>prefix</i>	R1226, C1244, C1245, C1246, C1247, C1249, C1250, C1251, C1252, R1234, C1256, R1226, C1244, C1245, C1246, C1247, C1249, C1250, C1251, C1252, R1234, C1256
R1230	<i>prefix-spec</i>	R1229, C1244, R1229, C1244
—	<i>primaries</i>	C1270
R701	<i>primary</i>	R702
R912	<i>print-stmt</i>	R214, C1284
R450	<i>private-components-stmt</i>	R433, C466, R433, C466
R433	<i>private-or-sequence</i>	R430, C437, R430, C437
R1213	<i>proc-attr-spec</i>	R1211
R453	<i>proc-binding-stmt</i>	R451
R447	<i>proc-component-attr-spec</i>	R446, C455, C456, C459, R446, C455, C456, C459
R446	<i>proc-component-def-stmt</i>	R440, C455, R440, C455
R741	<i>proc-component-ref</i>	R740, R742, R740, R742, R1221, R1223, R1221, R1223
R1214	<i>proc-decl</i>	R446, R1211, C1217, C1219, R1211, C1217, C1219
—	<i>proc-entity-name</i>	R546
R1212	<i>proc-interface</i>	R446, R1211, C1216, C1220, R1211, C1216, C1220
R1227	<i>proc-language-binding-spec</i>	C515, R1213, C1219, C1220, C1240, C1241, C1242, C1247, R1231, R1234, R1213, C1219, C1220, C1240, C1241, C1242, C1247, R1231, R1234

R1216	<i>proc-pointer-init</i>	R1214
R550	<i>proc-pointer-name</i>	R549, R564, C594, C597, R549, R564, C594, C597, R639, R740
R740	<i>proc-pointer-object</i>	R735
R742	<i>proc-target</i>	R462, C502, R462, C502, C546, R735, C733, R735, C732
—	<i>procedure-component-name</i>	C730, R741, C729
R1211	<i>procedure-declaration-stmt</i>	R207, C1213
R1221	<i>procedure-designator</i>	R1219, C1222, R1220, C1224, R1219, C1222, R1220, C1224
—	<i>procedure-entity-name</i>	R1214, C1216, R1214, C1216
—	<i>procedure-name</i>	R454, C468, C469, C470, R454, C468, C469, C470, R742, C732, R742, C731, R1206, C1207, C1208, C1209, R1217, C1218, R1221, C1225, R1223, C1232, C1233, R1237, R1238, C1260, C1261, R1206, C1207, C1208, C1209, C1218, R1221, C1225, R1223, C1232, C1233, R1237, R1238, C1260, C1261
R1206	<i>procedure-stmt</i>	R1202, C1204, C1208, C1209, R1202, C1204, C1208, C1209
—	<i>program-name</i>	R1102, R1103, C1102, R1102, R1103, C1102
R1102	<i>program-stmt</i>	R1101, C1102, R1101, C1102
R202	<i>program-unit</i>	R201
R547	<i>protected-stmt</i>	R212
R865	<i>query-spec</i>	R864, C844, R864, C844
R864	<i>query-stmt</i>	R214
R1005	<i>r</i>	R1003, C1003, C1004, R1012, R1003, C1003, C1004, R1012
R471	<i>rbracket</i>	R343, R444, R468, R444, R468, R503, R509, R523, R540, R552, R503, R509, R523, R540, R552, R624, R631, R624, R631
R910	<i>read-stmt</i>	R214, C912, C1285
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, R1109, R1112
—	<i>rep-char</i>	R423
—	<i>result-name</i>	C1238, R1231, C1267, C1238, R1231, C1267
R334	<i>result-token</i>	R333, C313, C314, R333, C313, C314
R1240	<i>return-stmt</i>	R214, C325, C824, C826, C824, C826, C1268
R925	<i>rewind-stmt</i>	R214, C1284
R1018	<i>round-edit-desc</i>	R1012
R548	<i>save-stmt</i>	R212
R549	<i>saved-entity</i>	R548
R103	<i>scalar-xyz</i>	C101
R620	<i>section-subscript</i>	R613, C614, C615, C622, R613, C614, C622
—	<i>section-subscript-list</i>	C615
R809	<i>select-case-stmt</i>	R808, C803, R808, C803
—	<i>select-construct-name</i>	R822, R823, R824, C819, R822, R823, R824, C819
R821	<i>select-type-construct</i>	R213, C817, C818, C819, C817, C818, C819
R822	<i>select-type-stmt</i>	R821, C813, C819, R821, C813, C819

R819	<i>selector</i>	R818, C808, R822, C811, C812, C813, C816, R818, C808, R822, C811, C812, C813, C816
R1237	<i>separate-module-subprogram</i>	R1108, C1260
R435	<i>sequence-stmt</i>	R433
R840	<i>shared-term-do-construct</i>	R839
R411	<i>sign</i>	R406, R409, R412, 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, R418, R419, R536, R1011, R1013, R1011, R1013
R412	<i>signed-real-literal-constant</i>	R418, R419, R418, R419, R536
R414	<i>significand</i>	R413
R630	<i>source-expr</i>	R627, C636, C637, R627, C636, C637
—	<i>special-character</i>	R301
R454	<i>specific-binding</i>	R453
R729	<i>specification-expr</i>	C404, C420, C404, C420, R513, R514, R513, R514, C1291
R204	<i>specification-part</i>	C471, C516, C558, C516, C558, R845, R1101, C1103, R1104, C1105, C1106, C1107, R1116, C1116, C1117, C1118, R1120, C1122, C1123, R1101, C1103, R1104, C1105, C1106, C1107, R1116, C1117, C1118, C1119, R1120, C1123, C1124, R1205, R1225, R1233, R1237, C1276, C1277, C1278, C1279, R1205, R1225, R1233, R1237, C1276, C1277, C1278, C1279
R212	<i>specification-stmt</i>	R207
R628	<i>stat-variable</i>	R627, R641, R627, R641, R858
R1242	<i>stmt-function-stmt</i>	R207, C1105, C1117, C1105, C1118, C1206
R856	<i>stop-code</i>	R855
R855	<i>stop-stmt</i>	R214, C325, C824, C826, C824, C826, C1286
R622	<i>stride</i>	R621, R755, C741, R755, C740
R614	<i>structure-component</i>	R532, C566, C567, C568, R532, C566, C567, C568, R603, R610, R632, R639, R603, R610, R632, R639
R460	<i>structure-constructor</i>	R536, C571, R536, C571, R701
R1116	<i>submodule</i>	R202
—	<i>submodule-name</i>	R1117, R1119, C1120, R1117, R1119, C1121
R1117	<i>submodule-stmt</i>	R1116, C1120, R1116, C1121
—	<i>subroutine-name</i>	C1203, R1234, R1236, C1259, C1203, R1234, R1236, C1259
R1234	<i>subroutine-stmt</i>	R1205, C1203, C1240, C1247, C1248, R1233, C1256, C1259, R1205, C1203, C1240, C1247, C1248, R1233, C1256, C1259
R1233	<i>subroutine-subprogram</i>	R203, R211, R203, R211, R1108
R619	<i>subscript</i>	C567, C621, R620, R621, C621, R620, R621, R755, C741, R755, C740
R621	<i>subscript-triplet</i>	R620, C625, R620, C625
R609	<i>substring</i>	R562, C592, R562, C592, R603
R611	<i>substring-range</i>	R609, R618, C623, R609, R618, C623
R1231	<i>suffix</i>	R1226, R1239, R1226, R1239
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, C841, R863, R865, R866, R857, C841, R863, R865, R866
R859	<i>sync-team-stmt</i>	R214
R552	<i>target-decl</i>	R551
R551	<i>target-stmt</i>	R212
R335	<i>token</i>	R334, C314, R334, C314
R432	<i>type-attr-spec</i>	R431, C432, R431, C432
R451	<i>type-bound-procedure-part</i>	R430, C440, R430, C440, C1504
R501	<i>type-declaration-stmt</i>	R207, C424, C425, C424, C425, C501
R823	<i>type-guard-stmt</i>	R821, C817, C818, C819, R821, C817, C818, C819
—	<i>type-name</i>	R431, C431, R434, C438, C476, R458, C488, R431, C431, R434, C438, C476, R458, 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, R430, C441, C442
R616	<i>type-param-inquiry</i>	R701
—	<i>type-param-name</i>	R431, R437, C441, C442, R431, R437, C441, C442, R616, C620, R616, C620, R701, C701, R701, C701
R459	<i>type-param-spec</i>	R458, C489, C490, C491, C493, R458, C489, C490, C491, C493
R401	<i>type-param-value</i>	C401, C402, C404, R420, R421, R422, C420, C421, C422, C423, C454, R459, C493, C401, C402, C404, R420, R421, R422, C420, C421, C422, C423, C454, R459, C493, C630
R402	<i>type-spec</i>	R469, C505, C506, C507, R469, C505, C506, C507, R626, C627, C628, C629, C630, C631, C635, R626, C627, C628, C629, C630, C631, C635, R823, C814, C815, C816, R823, C814, C815, C816
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, R633, R637, R738
R1115	<i>use-defined-operator</i>	R1111, C1111, C1115, R1111, C1112, C1116
—	<i>use-name</i>	R521, C559, R521, C559, R1111, R1113, C1114, R1111, R1113, C1115
R1109	<i>use-stmt</i>	R204
R1011	<i>v</i>	R1006, C1008, R1006, C1008
R553	<i>value-stmt</i>	R212
R601	<i>variable</i>	R530, C562, C564, R530, C562, C564, R604, R605, R606, R607, R608, R604, R605, R606, R607, R608, R734, C716, R736, C723, C724, R739, C726, C729, C730, C745, R734, C716, R736, C723, C724, R739, C726, R741, C729, C744, R819, C808, C811, C812, R819, C808, C811, C812, R915, R1223
R602	<i>variable-name</i>	R559, R562, R564, C594, C597, R559, R562, R564, C594, C597, C603, R610, R632, R639, C603, R610, R632, R639, R736, C722, R736, C722
R623	<i>vector-subscript</i>	R620, C624, R620, C624
R554	<i>volatile-stmt</i>	R212
R1007	<i>w</i>	R1006, C1006, C1007, C1008, R1006, C1006, C1007, C1008

R922	<i>wait-spec</i>	R921, C939, C940, R921, C939, C940
R921	<i>wait-stmt</i>	R214, C1284
R747	<i>where-assignment-stmt</i>	R743, R746, C734, R743, R746, C733
R746	<i>where-body-construct</i>	R744, C736, R744, C735
R744	<i>where-construct</i>	R213, R746, R756, R746, R756
——	<i>where-construct-name</i>	R745, R749, R750, R751, C735, R745, R749, R750, R751, C734
R745	<i>where-construct-stmt</i>	R744, C735, R744, C734
R743	<i>where-stmt</i>	R214, R746, R756, R746, R756
R911	<i>write-stmt</i>	R214, C913, C1285

# Annex E

(Informative)

## Index

In this index, 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

<, 152  
<=, 152  
>, 152  
>=, 152  
(, 154  
, 31, 42, 44, 46, 51, 91, 102, 124, 151, 234, 265,  
278, 283, 310, 329  
\*\*, 151  
+, 151  
-, 151  
.AND., 153, 154  
.EQ., 152  
.EQV., 153, 154  
.GE., 152  
.GT., 152  
.LE., 152  
.LT., 152  
.NE., 152  
.NEQV., 153, 154  
.NOT., 153, 154  
.OR., 153, 154  
.XOR., 153, 154  
/, 151  
//, 52, 151  
/=: 152  
;, 31  
==, 152  
&, 30  
&, 283

## A

abstract interface, **299**  
abstract interface block, **302**  
abstract type, **73**, **509**  
*ac-do-variable* (R475), 80, **80**, 81, 142, 144, 491,  
492  
*ac-implied-do* (R473), 80, **80**, 81, 145, 492  
*ac-implied-do-control* (R474), 80, **80**, 142–145  
*ac-spec* (R469), 80, **80**  
*ac-value* (R472), 80, **80**, 81  
access methods, **208**  
*access-id* (R521), 99, **99**, 100 657  
*access-spec* (R507), 33, 56, 57, 61, 63, 68–70, 83,  
86, **86**, 99, 307  
*access-stmt* (R520), 10, 99, **99**, 100  
**ACCESS= specifier**, 219, 249  
accessibility attribute, **86**  
accessibility statements, 99

*action-stmt* (R214), 11, **11**, 36, 177, 193  
*action-term-do-construct* (R836), 186, **186**  
**ACTION= specifier**, 219, 249  
actions, **208**  
active, **186**  
active combination of, **171**  
actual argument, **297**, **509**  
*actual-arg* (R1223), 310, **310**  
*actual-arg-spec* (R1222), 76, 309, 310, **310**  
*add-op* (R709), 27, 134, **134**  
*add-operand* (R705), 134, **134**, 155  
**ADVANCE= specifier**, 227  
advancing input/output statement, **212**  
affecter, **228**  
*alloc-opt* (R627), 124, **124**, 125  
allocatable array, **90**  
ALLOCATABLE attribute, **86**, 100  
ALLOCATABLE statement, 100  
allocatable variable, **509**  
*allocatable-decl* (R523), 100, **100**  
*allocatable-stmt* (R522), 10, **100**, 493  
ALLOCATE statement, **124**  
*allocate-co-array-spec* (R636), 124, **124**, 127  
*allocate-co-shape-spec* (R637), 124, **124**  
*allocate-object* (R632), 51, 52, 124, **124**, 125, 126, 128, 129, 508  
*allocate-shape-spec* (R633), 124, **124**  
*allocate-stmt* (R626), 11, **124**, 508  
allocated, **127**  
*allocation* (R631), 124, **124**, 126, 127  
*alphanumeric-character* (R302), 25, **25**, 27  
*alt-return-spec* (R1224), 193, 309, 310, **310**  
ancestor, **293**  
ancestor component, **74**  
*ancestor-module-name*, 294  
*and-op* (R719), 27, 136, **136**  
*and-operand* (R714), 136, **136**  
**APOSTROPHE (DELIM value)**, 288  
approximation methods, **47**  
*arg-name*, 62–64, 69  
*arg-token* (R344), 36, **36**  
argument, **509**  
argument association, 311, 492, **509**  
argument keyword, **22**, 311  
argument keywords, 340, 491  
arithmetic IF statement, 193  
*arithmetic-if-stmt* (R853), 11, 36, 186, 193, **193**  
array, **20**, 88–92, **119**, 119–122, **509**  
    assumed-shape, 90  
    assumed-size, 91  
    deferred-shape, 90  
    explicit-shape, 90  
array constructor, 80, **80**  
array element, **20**, 121, **509**  
array element order, **121**  
array elements, **119**  
array intrinsic assignment statement, **158**  
array pointer, **90**, **509**  
array section, **20**, **121**, **509**  
*array-constructor* (R468), 80, **80**, 81, 133  
*array-element* (R617), 101, 109, 115, 116, **120**  
*array-name*, 103, 493  
*array-section* (R618), 115, 120, **120**, 121  
*array-spec* (R510), 7, 84, 85, 88, 89, **89**, 91, 100, 103, 105, 111, 112  
ASCII, 424  
ASCII character set, **50**  
ASCII character type, **50**  
ASCII collating sequence, **53**  
assignment, 156–174  
    defined, 161  
    elemental array (FORALL), 168  
    intrinsic, 157  
    masked array (WHERE), 166  
    pointer, 162  
assignment statement, 156, **509**  
*assignment-stmt* (R734), 11, 157, **157**, 166, 169, 171, 335, 508  
ASSOCIATE construct, 180  
**ASSOCIATE construct**, 492  
associate name, **509**  
associate names, **180**  
*associate-construct* (R816), 11, **180**, 181  
*associate-construct-name*, 180, 181  
*associate-name*, 180–184, 491, 519  
*associate-stmt* (R817), 180, **180**, 181, 193  
associated, **22**  
associating entity, **502**  
association, **23**, **509**  
    argument, 311, 492  
    host, 493  
    name, 492  
    pointer, 496  
    sequence, 318  
    storage, 499  
    use, 493  
*association* (R818), 180, **180**  
association status  
    pointer, 496  
assumed type parameter, **42**  
assumed-shape array, **90**, **509**  
*assumed-shape-spec* (R515), 89, 90, **90**  
assumed-size array, **91**, **509**  
*assumed-size-spec* (R517), 89, **91**  
ASYNCHRONOUS attribute, **86**, 100  
ASYNCHRONOUS statement, 100  
*asynchronous-stmt* (R524), 10, **100**

**ASYNCHRONOUS= specifier**, 219, 227, 250

*attr-spec* (R502), 83, **83**, 84, 85, 105

attribute, **509**

accessibility, 86

ALLOCATABLE, 86, 100

ASYNCHRONOUS, 86, 100

BIND, 56, 59, 100

DIMENSION, 88, 103

EXTERNAL, 92

INTENT, 92, 103

INTRINSIC, 94

OPTIONAL, 94, 104

PARAMETER, 95, 104

POINTER, 95, 104

PRIVATE, 57, 86, 100

PROTECTED, 96, 105

PUBLIC, 57, 86, 100

SAVE, 101, 105

TARGET, 98, 105

VALUE, 99, 105

VOLATILE, 99, 105

attribute specification statements, 99–114

attributes, **83**, 85–99

automatic data object, **84**, **510**

## B

BACKSPACE statement, 246

*backspace-stmt* (R923), 11, **245**, 335

base object, **117**

base type, **73**, **510**

*basic-token* (R342), 36, **36**

*basic-token-sequence* (R341), 35, 36, **36**

belong, **510**

belongs, **175**, **510**

*binary-constant* (R426), 54, 55, **55**

BIND attribute, 56, 59, 100

BIND statement, 100

**BIND(C)**, 78, 87, 298, 300, 480, 482, 485

*bind-entity* (R526), 100, **100**

*bind-stmt* (R525), 10, **100**

binding, 70

binding label, **485**, **510**

binding name, **70**

*binding-attr* (R456), 69, **69**, 70

*binding-name*, 69, 70, 309, 325, 491, 521

*binding-private-stmt* (R452), 68, **68**, 70

bit model, 341

bits compatible, **44**, **45**

bits conversion, 160

bits editing, 265

bits intrinsic assignment statement, **158**

bits intrinsic operation, **138**

bits intrinsic operator, **138**

bits relational intrinsic operation, **138**

bits type, **54**

blank common, **111**

*blank-interp-edit-desc* (R1017), 261, **262**

**BLANK= specifier**, 219, 228, 250

block, **175**, **510**

*block* (R801), **175**, 176, 178, 180, 182, 185, 191, 192

block data program unit, **294**, **510**

*block-construct* (R845), 11, 191, **191**

*block-construct-name*, 191

*block-data* (R1120), 9, 294, **294**

*block-data-name*, 294, 295

*block-data-stmt* (R1121), 9, 294, **294**

*block-do-construct* (R826), 185, **185**

*block-stmt* (R846), 191, **191**

blocking, **203**

bounds, **510**

*bounds-remapping* (R738), 162, 163, **163**, 164

*bounds-spec* (R737), 162, 163, **163**, 164

*boz-literal-constant* (R425), 27, **54**

branch target statement, **193**

Branching, **193**

## C

C address, **473**

C character kind, **472**

C\_(C type), 471–483

C\_LOC function, 475

CALL statement, 309

*call-stmt* (R1220), 11, **309**, 310, 311

CASE construct, **178**

case index, **178**

*case-construct* (R808), 11, 178, **178**

*case-construct-name*, 178

*case-expr* (R812), 178, **178**

*case-selector* (R813), 178, **178**

*case-stmt* (R810), 178, **178**

*case-value* (R815), 178, **178**

*case-value-range* (R814), 178, **178**

CHAR intrinsic, 53

*char-constant* (R309), 27, **27**

*char-expr* (R725), 140, **140**, 144, 178

*char-initialization-expr* (R731), 87, 144, **144**, 178, 194, 225, 226

*char-length* (R422), 51, **51**, 61, 62, 83, 84

*char-literal-constant* (R423), 27, 32, **52**, 239, 261, 262

*char-selector* (R420), 46, 51, **51**

*char-string-edit-desc* (R1020), 260, **262**

*char-variable* (R606), 115, **115**, 214

**CHARACTER**, 50

character, **510**

*character* (R301), **25**

- character context, **29**
- CHARACTER declaration, 525
- character intrinsic assignment statement, **158**
- character intrinsic operation, **138**, 151
- character intrinsic operator, **138**
- character length parameter, **510**
- character literal constant, **52**
- character relational intrinsic operation, **138**
- character sequence type, **58**, 109, 501, 616
- character set, 25
- character storage unit, **499**, **510**
- character string, **50**, **510**
- character string edit descriptor, **260**
- character type, **50**, 50–54
- characteristics, **510**
- characteristics of a procedure, **298**
- child, **293**
- child data transfer statement, **237**, 236–241, 257
- CLASS**, 44
- class, **510**
- CLOSE statement, **223**
- close-spec* (R909), 223, **223**, 224
- close-stmt* (R908), 11, **223**, 335
- co-array, **21**, **510**
- co-array-name*, 103
- co-array-spec* (R511), 61–63, 84, 88, 89, **89**, 100, 103, 105
- co-bounds, **88**, **511**
- co-indexed object, **21**, **511**
- co-rank, **21**, **511**
- co-size, **21**
- co-subscript, **511**
- co-subscript* (R625), 21, 123, **123**
- collating sequence, 53, **53**, **511**
- collective subroutine, **339**, **511**
- comment, **30**, **31**
- common association, 113
- common block, **111**, 489, **511**, 556
- common block storage sequence, **112**
- COMMON statement, **111**, 111–114
- common-block-name*, 100, 105, 111, 293
- common-block-object* (R564), 111, **111**, 112, 293, 493
- common-stmt* (R563), 10, **111**, 493
- companion processor, **24**, **511**
- compatibility
  - FORTRAN 77, 4
  - Fortran 2003, 3
  - Fortran 90, 4
  - Fortran 95, 3
- COMPLEX**, 49
- complex part designator, **118**
- complex type, 49, **49**
- complex-literal-constant* (R417), 27, **49**
- complex-part-designator* (R615), 115, 118, **118**, 120
- component, **511**
- component keyword, **22**
- Component order, **67**
- component order, **511**
- component value, **75**
- component-array-spec* (R445), 61, 62, **62**, 63
- component-attr-spec* (R442), 61, **61**, 62, 64, 65
- component-data-source* (R462), 76, **76**, 77
- component-decl* (R443), 51, 61, **61**, 63, 65
- component-def-stmt* (R440), 61, **61**, 62
- component-dim-spec* (R444), 61, **62**
- component-initialization* (R448), 61, 65, **65**, 290
- component-name*, 61, 65
- component-part* (R439), 56, **61**, 68, 71
- component-spec* (R461), 76, **76**, 143
- components, 491
- computed GO TO statement, 193
- computed-goto-stmt* (R852), 11, 193, **193**
- concat-op* (R711), 27, 135, **135**
- concatenation, **52**
- conform, 157
- conformable, **21**, **511**
- conformance, 141, **511**
- connect team, **222**, **511**
- connect-spec* (R905), 217, **217**, 218, 222
- connected, **216**, **511**
- connected files, 216
- constant, **20**, 27, 41, **511**
  - character, 52
  - integer, 47
  - named, 104
- constant* (R305), 27, **27**, 102, 116, 133
- constant subobject, **20**
- constant-subobject* (R538), 102, **102**
- construct, **511**
- Construct association, 496
- construct association, **511**
- construct entity, **487**, **511**
- construct-name*, 191
- constructor
  - array, 80
  - derived-type, 76
  - structure, 76
- CONTAINS statement, 332
- contains-stmt* (R1241), 10, 68, 290, **332**
- contiguous, **87**
- CONTIGUOUS attribute, **87**
- contiguous-stmt* (R527), **101**
- continuation, 30, 31
- CONTINUE statement, 194
- continue-stmt* (R854), 11, 36, 185, 186, **194**
- control character, **25**

control edit descriptor, **260**  
 control edit descriptors, **275**  
 control information list, **224**  
 control mask, **511**  
*control-edit-desc* (R1012), 260, **261**  
 conversion  
   bits, 160  
   numeric, 160  
 correspond, **330**  
*critical-construct* (R848), 11, 192, **192**  
*critical-construct-name*, 192  
*critical-stmt* (R849), 192, **192**  
 current record, **211**  
 CYCLE statement, 185, 188  
*cycle-stmt* (R843), 11, 36, 186, 188, **188**

## D

*d* (R1009), 261, **261**, 267–270, 273, 282  
 data, **512**  
 data edit descriptor, **260**  
 data edit descriptors, 264–274  
 data entity, **19**, **512**  
 data object, **19**, **512**  
 data object reference, **23**  
 data pointer, **22**  
 DATA statement, 101, 504  
 data transfer, 234  
 data transfer input statement, **207**  
 data transfer output statements, **207**  
 data transfer statements, 224  
 data type, *see* type, **512**  
*data-component-def-stmt* (R441), 61, **61**, 63  
*data-edit-desc* (R1006), 260, **260**  
*data-i-do-object* (R532), 101, **101**, 102  
*data-i-do-variable* (R533), 101, **101**, 102, 144, 491, 492  
*data-implied-do* (R531), 101, **101**, 102, 144, 492  
*data-pointer-component-name*, 163  
 data-pointer-initialization compatible, **65**  
*data-pointer-object* (R736), 162, 163, **163**, 164, 171, 172, 356, 508  
*data-ref* (R612), 117, **117**, 118, 120, 163, 228, 309, 311, 317, 325  
*data-stmt* (R528), 10, **101**, 301, 334, 493  
*data-stmt-constant* (R536), 102, **102**, 103  
*data-stmt-object* (R530), 101, **101**, 102  
*data-stmt-repeat* (R535), 102, **102**  
*data-stmt-set* (R529), 101, **101**  
*data-stmt-value* (R534), 101, 102, **102**  
*data-target* (R739), 76, 77, 96, 162, 163, **163**, 164, 171, 319, 334, 356, 499  
 datum, **512**  
*dealloc-opt* (R641), 128, 129, **129**, 131  
 DEALLOCATE statement, **128**

*deallocate-stmt* (R640), 11, **128**, 508  
 decimal symbol, **264**, **512**  
*decimal-edit-desc* (R1019), 261, **262**  
**DECIMAL= specifier**, 219, 228, 250  
 declaration, **23**  
*declaration-construct* (R207), 10, **10**  
*declaration-type-spec* (R403), 43, **43**, 44, 51, 61, 62, 83, 84, 106, 142, 307, 326, 329  
 declarations, 83–114  
 declared type, **44**, **512**  
 default bits, **54**  
 default character, **51**  
 default complex, **49**  
 default initialization, **512**  
 default integer, **47**  
 default logical, **54**  
 default real, **48**  
*default-char-expr* (R726), 140, **140**, 217–221, 223–229  
*default-char-variable* (R607), 115, **115**, 124, 218, 248–254  
 default-initialized, **66**, **512**  
*default-logical-variable* (R605), 115, **115**, 248–252  
 deferred binding, **70**, **512**  
 deferred type parameter, **42**, **512**  
 deferred-shape array, **90**  
*deferred-shape-spec* (R516), 62, 89, 91, **91**, 104  
 definable, **512**  
*define-macro-stmt* (R315), 33, **33**, 494  
 defined, **23**, **512**  
 defined assignment, 305  
 defined assignment statement, **161**, **512**  
 defined binary operation, **139**  
 defined elemental assignment statement, **161**  
 defined elemental operation, **139**  
 defined operation, **138**, 304, **512**  
 defined unary operation, **138**  
*defined-binary-op* (R723), 28, 136, **136**, 139, 155, 292  
*defined-operator* (R311), **28**, 69, 292, 301  
*defined-unary-op* (R703), 28, 134, **134**, 138, 139, 154, 292  
 definition, **23**  
 definition of variables, 503  
 deleted feature, **512**  
 deleted features, **7**  
**DELIM= specifier**, 219, 228, 250  
 Delimiters, **29**  
 derived type, **19**, **512**  
 derived type determination, 58  
 derived types, 56–78  
 derived-type intrinsic assignment statement, **158**

derived-type type specifier, 44  
*derived-type-def* (R430), 10, 44, **56**, 57, 60  
*derived-type-spec* (R458), 43, 44, 51, **75**, 76, 182, 237, 238, 491  
*derived-type-stmt* (R431), 56, **56**, 57, 60, 493  
 descendant, **293**  
 designator, **22**, **512**  
*designator* (R603), 65, 102, 115, **115**, 118, 119, 133, 169  
*designator*, 133  
*digit*, 5, 6, 25, **26**, 28, 47, 55, 280  
*digit-string* (R410), 47, **47**, 48, 49, 266, 267, 271  
*digit-string*, 47  
 digits, **25**  
 DIMENSION attribute, **88**, 103  
 DIMENSION statement, 103  
*dimension-decl* (R540), 103, **103**  
*dimension-spec* (R509), **89**  
*dimension-stmt* (R539), 10, **103**, 493  
 direct access, 210  
 direct access input/output statement, **229**  
**DIRECT= specifier**, 250  
 disassociated, **22**, **513**  
 distinguishable, **490**  
 DO construct, 185  
 DO statement, 185  
 DO termination, **186**  
 DO WHILE statement, 185  
*do-block* (R832), 185, **185**, 186, 187  
*do-body* (R837), 186, **186**  
*do-construct* (R825), 11, **185**, 188, 191  
*do-construct-name*, 185, 188  
*do-stmt* (R827), 185, **185**, 193, 508  
*do-term-action-stmt* (R838), 36, 186, **186**, 188, 193  
*do-term-shared-stmt* (R842), 186, **186**, 187, 188, 193  
*do-variable* (R831), 80, 101, 185, **185**, 187, 230, 255–257, 280, 504, 506, 508, 548  
**DOUBLE PRECISION**, 48  
 double precision real, **48**  
*dtio-generic-spec* (R1208), 69, 75, 236–239, 243, 301, **301**, 304, 490  
*dtv-type-spec* (R920), **237**  
 dummy argument, **297**, **513**  
 dummy arguments  
     restrictions, 319  
 dummy array, **513**  
 dummy data object, **513**  
 dummy procedure, **298**, **513**  
*dummy-arg* (R1235), 329, **329**, 330, 331  
*dummy-arg-name* (R1228), 103–105, 326, **326**, 329, 333, 493  
 dynamic type, **44**, **513**

## E

*e* (R1010), 261, **261**, 268, 269, 273, 274, 282  
 edit descriptor, **260**  
 edit descriptors, *see* format descriptors  
 effective item, **231**, **513**  
 effective position, **490**  
 element array assignment (FORALL), 168  
 element sequence, **318**  
 elemental, **21**, **297**, **513**  
 elemental intrinsic function, **339**  
 elemental operation, **141**  
 elemental procedure, **335**  
*else-if-stmt* (R804), 176, **176**  
*else-stmt* (R805), 176, **176**  
*elsewhere-stmt* (R750), 166, **166**  
**ENCODING= specifier**, 219, 250  
 END statement, 15, **15**  
*end-associate-stmt* (R820), 180, 181, **181**, 193  
*end-block-data-stmt* (R1122), 10, 15, 294, **294**  
*end-block-stmt* (R847), 191, **191**  
*end-critical-stmt* (R850), 192, **192**  
*end-do* (R833), 185, **185**, 186, 188  
*end-do-stmt* (R834), 185, **185**, 193  
*end-enum-stmt* (R467), 78, **79**  
*end-forall-stmt* (R758), 168, 169, **169**  
*end-function-stmt* (R1232), 9, 11, 15, 36, 177, 186, 301, 326, 327, **327**, 332  
*end-if-stmt* (R806), 176, **176**, 193  
*end-interface-stmt* (R1204), 300, 301, **301**  
*end-macro-stmt* (R336), 33, **35**  
*end-module-stmt* (R1106), 9, 15, 290, **290**  
 end-of-file condition, **255**  
 end-of-record condition, **255**  
*end-program-stmt* (R1103), 9, 11, 15–17, 36, 177, 186, 194, 195, 289, **289**  
*end-select-stmt* (R811), 178, **178**, 179, 193  
*end-select-type-stmt* (R824), 182, 183, **183**, 184, 193  
*end-sep-subprogram-stmt* (R1238), 15, 330, **330**  
*end-submodule-stmt* (R1119), 9, 15, 294, **294**  
*end-subroutine-stmt* (R1236), 9, 11, 15, 36, 177, 186, 301, 329, **329**, 332  
*end-type-stmt* (R434), 56, **57**  
*end-where-stmt* (R751), 166, **166**  
**END= specifier**, 255  
 endfile record, **208**  
 ENDFILE statement, 208, 246  
*endfile-stmt* (R924), 11, **245**, 335  
 ending point, **116**  
 entity, **513**  
*entity-decl* (R503), 51, 83, 84, **84**, 85, 97, 143, 144, 493  
*entity-name*, 100, 105  
 ENTRY statement, 330, **330**



*entry-name*, 326, 330, 331, 489  
*entry-stmt* (R1239), 10, 290, 301, 330, **330**, 331, 489, 493  
*enum-def* (R463), 10, **78**, 79  
*enum-def-stmt* (R464), 78, **78**  
enumeration, 78  
enumerator, 78  
*enumerator* (R466), 78, 79, **79**  
*enumerator-def-stmt* (R465), 78, **78**  
**EOR= specifier**, 256  
*equiv-op* (R721), 27, 136, **136**  
*equiv-operand* (R716), 136, **136**  
EQUIVALENCE statement, **108**, 108–111  
*equivalence-object* (R562), 109, **109**, 111, 293  
*equivalence-set* (R561), 109, **109**, 110  
*equivalence-stmt* (R560), 10, **109**, 493  
**ERR= specifier**, 255  
*errmsg-variable* (R629), 124, **124**, 125, 128, 129, 198, 508  
**ERRMSG= specifier**, 128  
ERROR\_UNIT, 215, 439  
evaluation

- optional, 146
- operations, 145
- parentheses, 147

executable construct, **513**  
executable constructs, 175  
executable statement, **14**, **513**  
*executable-construct* (R213), 10, **11**, 14, 331  
execution control, 175–195  
execution cycle, **187**  
*execution-part* (R208), 9, **10**, 11, 289, 326, 327, 329, 330  
*execution-part-construct* (R209), 10, **10**, 175, 186  
exist, **209**  
**EXIST= specifier**, 250  
*exit-stmt* (R844), 11, 36, 186, 191, **191**  
*expand-stmt* (R338), 33, **35**, 36  
explicit, **299**  
explicit formatting, 259–278  
explicit initialization, **85**, 101, **513**  
explicit interface, 299, **513**  
explicit-shape array, **90**, **513**  
*explicit-shape-spec* (R512), 62, 85, 89, 90, **90**, 91, 92, 111  
*exponent* (R416), 48, **49**  
*exponent-letter* (R415), 48, 49, **49**  
*expr* (R722), 6, 72, 76, 77, 80, 115, 124, 133, 134, 136, **136**, 139, 140, 144, 157–164, 168, 171, 181, 230, 310, 333, 334  
expression, **133**, **513**  
expressions, **20**, 133–155  
extended type, **73**, **513**

*extended-intrinsic-op* (R312), 28, **28**  
extensible type, **73**, **513**  
extension operation, **139**, 154  
extension operator, **139**  
extension type, **73**, **513**  
extent, **21**, **514**  
EXTERNAL attribute, **92**  
external file, **208**, **514**  
external linkage, **514**  
external procedure, **13**, **297**, **514**  
EXTERNAL statement, **306**  
external subprogram, **12**, **514**  
external unit, **214**, **514**  
*external-name*, 306  
*external-stmt* (R1210), 10, **306**, 493  
*external-subprogram* (R203), 9, **9**, 330

## F

field, **262**  
field width, **262**  
file, **514**  
file access, 209  
file connection, 214  
file connection statements, **207**  
file inquiry, 248  
file inquiry statement, **207**  
file position, 211  
file positioning statements, **207**, 245  
file storage unit, **213**, **514**  
*file-name-expr* (R906), 218, **218**, 220, 248, 249, 251  
*file-unit-number* (R902), 214, **214**, 215, 217, 218, 223–226, 239, 244, 245, 247–249, 335, 440  
**FILE= specifier**, 220, 249  
files

- connected, 216
- external, 208
- internal, 214
- preconnected, 217

final subroutine, **514**  
final subroutines, **71**  
*final-binding* (R457), 69, **71**  
*final-subroutine-name*, 71  
finalizable, **71**, **514**  
finalization, 71, **514**  
finalized, **71**  
fixed source form, 31, **31**  
FLUSH statement, 247  
*flush-spec* (R928), 247, **247**  
*flush-stmt* (R927), 11, **247**, 335  
**FMT= specifier**, 226  
FORALL construct, 168

*forall-assignment-stmt* (R757), 169, **169**, 171–173, 335  
*forall-body-construct* (R756), 168, 169, **169**, 171, 173  
*forall-construct* (R752), 11, **168**, 169, 170, 172, 173  
*forall-construct-name*, 168, 169  
*forall-construct-stmt* (R753), 168, **168**, 169, 193  
*forall-header* (R754), 168, **168**, 173, 174, 185  
*forall-stmt* (R759), 11, 169, 173, **173**  
*forall-triplet-spec* (R755), 168, **168**, 169, 171, 173  
**FORM= specifier**, 220, 251  
*format* (R914), 224–226, **226**, 227, 234, 259, 260  
format control, **262**  
format descriptors  
  /, 276  
  :, 276  
  A, 272  
  BN, 277  
  BZ, 277  
  control edit descriptors, 275  
  D, 268  
  data edit descriptors, 264–274  
  E, 268  
  EN, 268  
  ES, 269  
  F, 267  
  G, 272, 273  
  I, 265  
  L, 272  
  P, 277  
  S, 276  
  SP, 276  
  SS, 276  
  TL, 275  
  TR, 275  
  X, 276  
**FORMAT** statement, 226, 259, 630  
*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, **226**  
formatted record, **207**  
**FORMATTED= specifier**, 251  
formatting  
  explicit, 259–278  
  list-directed, 236, 278–282  
  namelist, 236, 283–288  
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, 29, **29**  
function, **12**, **514**  
function reference, **20**, 322  
function result, **514**  
**FUNCTION** statement, 326  
function subprogram, **326**, **514**  
*function-name*, 84, 301, 302, 326, 327, 331, 333, 489, 493  
*function-reference* (R1219), 77, 84, 133, **309**, 311, 322  
*function-stmt* (R1226), 9, 301, 302, 326, **326**, 327, 489, 493  
*function-subprogram* (R1225), 9, 10, 290, **326**, 330

## G

generic identifier, **304**, **514**  
generic interface, 70, **303**, **514**  
generic interface block, **302**, **514**  
generic name, **304**  
Generic names, **340**  
generic procedure references, 489  
*generic-binding* (R455), 69, **69**, 70, 292  
*generic-name*, 69, 70, 301, 491, 494  
*generic-spec* (R1207), 69, 70, 75, 99, 138, 139, 162, 291, 292, 300, 301, **301**, 304, 491, 494  
global entities, **487**  
global entity, **514**  
global identifier, **487**  
**GO TO** statement, 193  
*goto-stmt* (R851), 11, 36, 186, 193, **193**  
graphic character, **25**

## H

*hex-constant* (R428), 55, **55**  
*hex-digit* (R429), 55, **55**, 271  
*hex-digit-string* (R1021), 271, **271**  
host, **12**, **13**, **514**  
host association, **493**, **514**  
host instance, **164**  
host scoping unit, **12**, **514**

## I

ICHAR intrinsic, 53  
**ID= specifier**, 228, 251  
IEEE\_, 350, 443–469  
**IF** construct, 176, **176**  
**IF** statement, **176**, 177  
*if-construct* (R802), 11, 176, **176**  
*if-construct-name*, 176

- if-stmt* (R807), 11, 36, 177, **177**
- if-then-stmt* (R803), 176, **176**, 193
- imag-part* (R419), 49, **50**
- image, **17**, **514**
- image control, **196**
- image index, **18**, **515**
- image selector, **123**
- image-selector* (R624), 21, 117, **123**, 511
- image-set* (R862), 201, **201**, 202, 203
- image-team* (R860), 199, **199**, 202, 218, 222
- IMAGE\_TEAM, 438
- imaginary part, **49**
- implicit, **299**
- implicit interface, 309, **515**
- IMPLICIT statement, 106, **106**
- implicit-part* (R205), 10, **10**
- implicit-part-stmt* (R206), 10, **10**
- implicit-spec* (R556), 106, **106**
- implicit-stmt* (R555), 10, **106**
- implied-shape array, **92**
- implied-shape-spec* (R518), 89, 92, **92**
- import-name*, 301
- import-name-list*, 302
- import-stmt* (R1209), 10, **301**
- inactive, **186**
- INCLUDE**, 32
- INCLUDE line, **32**
- index-name*, 168–174, 187, 188, 491, 492, 505, 507
- inherit, **515**
- inheritance associated, **74**
- inheritance association, 502, **515**
- inherited, **73**
- initial point, **211**
- initial-data-target* (R449), 65, **65**, 84, 85, 102, 103
- initial-proc-target* (R1217), 65, **307**, 308
- initialization, 65, 84, 85, 503, 610
- initialization* (R505), 84, **84**, 85
- initialization expression, **143**
- initialization target, **65**
- initialization-expr* (R730), 65, 84, 85, 92, 95, 104, 144, **144**
- inner-shared-do-construct* (R841), 186, **186**
- Input statements, **207**
- input-item* (R915), 224, 225, 230, **230**, 243, 257, 508
- input/output editing, 259–288
- input/output list, 229
- input/output statements, 207–257
- INPUT\_UNIT, 215, 440
- inquire by file, **248**
- inquire by output list, **248**
- inquire by unit, **248**
- INQUIRE statement, 248
- inquire-spec* (R930), 248, **248**, 249, 254, 257
- inquire-stmt* (R929), 11, **248**, 335
- inquiry function, **339**, **515**
- inquiry, type parameter, 119
- instance, **329**
- instance of a subprogram, **515**
- int-constant* (R308), 27, **27**, 102
- int-constant-name*, 47
- int-constant-subobject* (R537), 102, **102**
- int-expr* (R727), 15, 42, 80, 116, 120, 123, 124, 140, **140**, 142–145, 168, 178, 185, 187, 193, 201, 214, 215, 218, 225, 230, 232, 244, 248, 332
- int-initialization-expr* (R732), 46, 51, 60, 79, 101, 144, **144**, 178, 194
- int-literal-constant* (R407), 27, 47, **47**, 51, 260, 261
- int-variable* (R608), 115, **115**, 124, 218, 223, 225, 244, 245, 247–249, 251–256
- int-variable-name*, 185
- INTEGER**, 47
- integer constant, 47
- integer editing, 265
- integer model, 342
- integer type, **46**, **46**
- INTENT**, 341
- intent, **515**
- INTENT attribute, **92**, 103
- INTENT statement, 103
- intent-spec* (R519), 83, **92**, 103, 307
- intent-stmt* (R541), 10, **103**
- interface, **299**
  - (procedure), 299
  - explicit, 299
  - generic, 303
  - implicit, 309
- interface block, **13**, **515**
- interface body, **13**, **515**
- interface of a procedure, **515**
- interface-block* (R1201), 10, **300**, 301
- interface-body* (R1205), 300, 301, **301**, 493
- interface-name* (R1215), 307, **307**
- interface-name*, 69, 70
- interface-specification* (R1202), 300, **300**
- interface-stmt* (R1203), 300, **300**, 301, 304, 494
- internal file, **515**
- internal files, 214
- internal procedure, **13**, **297**, **515**
- internal subprogram, **12**, **515**
- internal unit, **214**
- internal-file-variable* (R903), 214, **214**, 226, 508
- internal-subprogram* (R211), 10, **10**

*internal-subprogram-part* (R210), 9, **10**, 289, 326, 327, 329, 330, 334  
 interoperable, 476, **477**, **480–482**, 483, **515**  
 intrinsic, **23**, **516**  
   elemental, 339  
   inquiry function, 339  
   transformational, 339  
 intrinsic assignment statement, **157**  
 INTRINSIC attribute, **94**  
 intrinsic binary operation, **137**  
 intrinsic module, **290**  
 intrinsic operation, **137**  
 intrinsic operations, 150–153  
   logical, 54  
 intrinsic procedure, **297**  
 intrinsic procedures, 350, 437  
 INTRINSIC statement, **309**  
 intrinsic type, **18**  
 intrinsic types, 46–54  
 intrinsic unary operation, **137**  
*intrinsic-operator* (R310), **27**, 28, 134, 136–139, 304  
*intrinsic-procedure-name*, 309, 494  
*intrinsic-stmt* (R1218), 10, **309**, 494  
*intrinsic-type-spec* (R404), 43, 44, **46**, 51  
 invoke, **516**  
*io-control-spec* (R913), 224, 225, **225**, 226, 239, 257, 258  
*io-implied-do* (R917), 230, **230**, 231, 232, 235, 257, 258, 504, 506, 508, 548  
*io-implied-do-control* (R919), 230, **230**, 232  
*io-implied-do-object* (R918), 230, **230**  
*io-unit* (R901), 214, **214**, 225, 226, 335  
*iomsg-variable* (R907), 218, **218**, 223, 225, 244, 245, 247, 248, 255–257, 505  
**IOMSG= specifier**, 257  
**IOSTAT= specifier**, 256  
 ISO 10646, 424  
 ISO 10646 character set, **50**  
 ISO 10646 character type, **50**  
 ISO\_C\_BINDING module, 471  
 ISO\_FORTRAN\_ENV module, 214, 215, 437  
 iteration, **188**  
 iteration count, **37**, **187**

## K

*k* (R1013), 261, **261**, 268, 273, 277  
 keyword, **22**, 311, **516**  
*keyword* (R215), 22, **23**, 75, 76, 310  
 kind, **47**, **49**, **50**, **54**  
 KIND intrinsic, 47–50, 54  
 kind type parameter, **18**, 42, 47, 49, 50, 54, **516**  
*kind-param* (R408), 47, **47**, 48, 49, 51, 52, 54, 55

*kind-selector* (R405), 7, 46, **46**, 60

## L

label, **516**  
*label* (R313), 28, **28**, 185, 193, 218, 223–226, 244, 245, 247–249, 255, 256, 310  
*label-do-stmt* (R828), 185, **185**, 186  
*language-binding-spec* (R508), 83, **87**, 100, 326  
*lbracket* (R470), 36, 61, 62, 80, **80**, 84, 89, 100, 103, 105, 123, 124  
 left tab limit, **275**  
 length, **50**  
 length of a character string, **516**  
 length type parameter, 42  
*length-selector* (R421), 7, 51, **51**  
*letter*, 25, **25**, 27, 28, 106, 134, 136  
*letter-spec* (R557), 106, **106**  
 letters, **25**  
*level-1-expr* (R702), 134, **134**, 155  
*level-2-expr* (R706), 134, **134**, 135, 155  
*level-3-expr* (R710), 135, **135**  
*level-4-expr* (R712), 135, **135**, 136  
*level-5-expr* (R717), 136, **136**  
 lexical token, **516**  
 Lexical tokens, **26**  
 line, **29**, **516**  
 linkage association, **516**  
 list-directed formatting, 236, 278–282  
 list-directed input/output statement, **227**  
 literal constant, **20**, 116, **516**  
*literal-constant* (R306), 27, **27**  
 local entity, **516**  
 local identifier, **487**  
 local identifiers, 488  
 local variable, **19**, **516**  
*local-defined-operator* (R1114), 291, 292, **292**  
*local-name*, 291–293  
**LOGICAL**, 54  
 logical intrinsic assignment statement, **158**  
 logical intrinsic operation, **138**  
 logical intrinsic operations, 54, 153  
 logical intrinsic operator, **138**  
 logical type, 54, **54**  
*logical-expr* (R724), 139, **139**, 144, 166, 176–178, 185, 187, 188  
*logical-initialization-expr* (R733), 144, **144**, 178  
*logical-literal-constant* (R424), 27, **54**, 134, 136  
*logical-variable* (R604), 115, **115**, 203  
 loop, **185**  
*loop-control* (R830), 185, **185**, 186, 187, 190  
 low-level syntax, **26**  
*lower-bound* (R513), 90, **90**, 91, 92  
*lower-bound-expr* (R634), 124, **124**, 127, 163

## M

*m* (R1008), 260, 261, **261**, 266, 271, 272  
*macro-actual-arg* (R339), 35, 36, **36**  
*macro-actual-arg-value* (R340), 36, **36**  
*macro-attribute* (R316), 33, **33**  
*macro-body-block* (R321), 33, **33**, 34  
*macro-body-construct* (R322), 33, **33**, 34  
*macro-body-stmt* (R333), 34, **34**, 35  
*macro-condition* (R332), 34, **34**  
*macro-declaration-stmt* (R317), 33, **33**  
*macro-definition* (R314), 10, 33, **33**, 34  
*macro-do-construct* (R323), 34, **34**  
*macro-do-limit* (R325), 34, **34**  
*macro-do-stmt* (R324), 34, **34**, 37  
*macro-do-variable-name*, 34, 37  
*macro-dummy-arg-name*, 33  
*macro-dummy-arg-name-list*, 33  
*macro-dummy-name*, 36  
*macro-else-if-stmt* (R329), 34, **34**  
*macro-else-stmt* (R330), 34, **34**  
*macro-end-do-stmt* (R326), 34, **34**  
*macro-end-if-stmt* (R331), 34, **34**  
*macro-expr* (R337), 33–35, **35**, 37  
*macro-if-construct* (R327), 34, **34**  
*macro-if-then-stmt* (R328), 34, **34**  
*macro-local-variable-name*, 33  
*macro-local-variable-name-list*, 33  
*macro-name*, 33, 35  
*macro-optional-decl-stmt* (R319), **33**  
*macro-type-declaration-stmt* (R318), 33, **33**  
*macro-type-spec* (R320), 33, **33**, 36  
 main program, 12, **289**, **516**  
*main-program* (R1101), 9, 289, **289**  
 many-one array section, **122**, **516**  
*mask-expr* (R748), 166, **166**, 167–173  
 masked array assignment, **166**  
 masked array assignment (WHERE), 166  
*masked-elsewhere-stmt* (R749), 166, **166**, 167, 172  
 model  
     bit, 341  
     integer, 342  
     real, 342  
 module, **13**, **290**, **516**  
*module* (R1104), 9, **290**  
 module procedure, **13**, **298**, **516**  
 module procedure interface, **302**, **517**  
 module procedure interface body, **302**  
 module reference, **23**, **291**  
 module subprogram, **12**, **517**  
*module-name*, 290–292, 493  
*module-nature* (R1110), 291, **291**, 292  
*module-stmt* (R1105), 9, 290, **290**  
*module-subprogram* (R1108), 10, 290, **290**, 330

*module-subprogram-part* (R1107), 9, 70, 74, 290, **290**, 294, 563  
*mult-op* (R708), 27, 134, **134**  
*mult-operand* (R704), 134, **134**, 155

## N

*n* (R1015), 261, **261**  
 name, **22**, **517**  
*name* (R304), 6, 23, 27, **27**, 84, 105, 115, 183, 234, 307, 326  
 name association, **492**, **517**  
 name-value subsequences, **283**  
**NAME= specifier**, 251  
 named, **517**  
 named common block, **111**  
 named constant, **20**, 95, 104, 116, **517**  
 named file, **208**  
*named-constant* (R307), 27, **27**, 32, 50, 79, 104, 493  
*named-constant-def* (R544), 104, **104**, 493  
**NAMED= specifier**, 251  
 namelist formatting, 236, 283–288  
 namelist input/output statement, **227**  
 NAMELIST statement, 108, **108**  
*namelist-group-name*, 108, 225–227, 233–235, 259, 283, 288, 293, 494, 508  
*namelist-group-object* (R559), 108, **108**, 234, 236, 243, 257, 283, 284, 293  
*namelist-stmt* (R558), 10, **108**, 494, 508  
 Names, **27**  
 NaN, 350, 448, **517**  
*nested-token-sequence* (R343), 36, **36**  
 next record, **211**  
**NEXTREC= specifier**, 251  
**NML= specifier**, 227  
 nonadvancing input/output statement, **212**  
*nonblock-do-construct* (R835), 185, **186**  
**NONE (DELIM value)**, 287  
 nonexecutable statement, **517**  
 Nonexecutable statements, **14**  
 nonintrinsic module, **290**  
*nonlabel-do-stmt* (R829), 185, **185**  
 normal, **448**  
*not-op* (R718), 27, 136, **136**  
*notify-stmt* (R863), 11, **202**  
*null-init* (R506), 65, 84, **84**, 85, 102, 103, 307, 308  
 NULLIFY statement, **128**  
*nullify-stmt* (R638), 11, **128**, 508  
**NUMBER= specifier**, 252  
 numeric conversion, 160  
 numeric editing, 265  
 numeric intrinsic assignment statement, **158**  
 numeric intrinsic operation, **137**

numeric intrinsic operations, 150  
 numeric intrinsic operator, **137**  
 numeric relational intrinsic operation, **138**  
 numeric sequence type, **58**, 109, 501, 616  
 numeric storage unit, **499**, **517**  
 numeric type, **517**  
 numeric types, **46**  
*numeric-expr* (R728), 48, 140, **140**, 193

## O

object, *see* data object, **19**, **517**  
 object designator, **22**, **517**  
*object-name* (R504), 84, **84**, 100, 101, 104, 105, 115, 493  
 obsolescent feature, **517**  
 obsolescent features, **7**  
*octal-constant* (R427), 55, **55**  
*only* (R1112), 291, **291**, 292  
*only-use-name* (R1113), 291, **291**, 292  
 OPEN statement, 217, **217**  
*open-stmt* (R904), 11, **217**, 335  
**OPENED= specifier**, 252  
 operand, **517**  
 operands, **24**  
 operation, **517**  
 operations, 41
 

- character intrinsic, 151
- logical intrinsic, 153
- numeric intrinsic, 150
- relational intrinsic, 152

 operator, **24**, **517**  
 operator precedence, 154  
 operators, 27  
 OPTIONAL attribute, **94**, 104  
 optional dummy argument, 319  
 OPTIONAL statement, 104  
*optional-stmt* (R542), 10, **104**  
*or-op* (R720), 27, 136, **136**  
*or-operand* (R715), 136, **136**  
*outer-shared-do-construct* (R839), 186, **186**  
 Output statements, **207**  
*output-item* (R916), 224, 225, 230, **230**, 243, 248  
 OUTPUT\_UNIT, 215, 440  
 override, 74, **517**  
 overrides, **65**

## P

**PAD= specifier**, 220, 228, 252  
 PARAMETER, 20  
 PARAMETER attribute, **95**, 104  
 PARAMETER statement, 104, **104**  
*parameter-stmt* (R543), 10, **104**, 493  
 parent, **293**  
 parent component, **74**, **517**

parent data transfer statement, **237**, 236–241, 257  
 parent type, **73**, **517**  
*parent-identifier* (R1118), 293, 294, **294**  
*parent-string* (R610), 88, 116, **116**  
*parent-submodule-name*, 294  
*parent-type-name*, 56, 57  
 parentheses, 147  
*part-name*, 117, 118, 120  
*part-ref* (R613), 88, 101, 109, 117, **117**, 118, 120, 121, 317  
 partially [storage] associated, **501**  
**PASS attribute**, 311  
 passed-object dummy argument, **64**, **518**  
**PENDING= specifier**, 252  
 pointer, **22**, **518**  
 pointer assignment, 162, **518**  
 pointer assignment statement, **518**  
 pointer associated, **518**  
 pointer association, 496, **518**  
 pointer association context, 508  
 pointer association status, **496**  
 POINTER attribute, **95**, 104  
 POINTER statement, 104  
*pointer-assignment-stmt* (R735), 11, 96, **162**, 169, 171, 334, 508  
*pointer-decl* (R546), 104, **104**  
*pointer-object* (R639), 128, **128**, 508  
*pointer-stmt* (R545), 10, **104**, 493  
 polymorphic, **44**, **518**  
**POS= specifier**, 229, 252  
 position, **209**  
*position-edit-desc* (R1014), 261, **261**  
*position-spec* (R926), 245, **245**  
**POSITION= specifier**, 220, 253  
 positional arguments, 340  
*power-op* (R707), 27, 134, **134**  
 pre-existing, **502**  
 precedence of operators, 154  
 preceding record, **211**  
 preconnected, **518**  
 preconnected files, 217  
 Preconnection, **217**  
*prefix* (R1229), 326, **326**, 327, 329  
*prefix-spec* (R1230), 326, **326**, 327–329, 333, 335  
 present, **319**  
 present (dummy argument), 319  
 PRESENT intrinsic, 94  
*primaries*, 333  
 primary, 133  
*primary* (R701), 133, **133**, 134  
 PRINT statement, **224**  
*print-stmt* (R912), 11, **224**, 335  
 PRIVATE attribute, 57, 86

PRIVATE statement, 100  
*private-components-stmt* (R450), 57, 67, **67**, 68  
*private-or-sequence* (R433), 56, 57, **57**  
*proc-attr-spec* (R1213), 306, 307, **307**  
*proc-binding-stmt* (R453), 68, **69**, 70  
*proc-component-attr-spec* (R447), 62, **62**, 63  
*proc-component-def-stmt* (R446), 61, **62**, 63  
*proc-component-ref* (R741), 163, **163**, 164, 309, 310  
*proc-decl* (R1214), 62, 65, 306, 307, **307**, 308  
*proc-entity-name*, 104, 308  
*proc-interface* (R1212), 62, 306, 307, **307**  
*proc-language-binding-spec* (R1227), 86, 307, 326, **326**, 327, 329, 332, 482  
*proc-pointer-init* (R1216), 307, **307**  
*proc-pointer-name* (R550), 105, **105**, 111, 112, 128, 163, 493  
*proc-pointer-object* (R740), 162, **163**, 164, 165, 171, 172, 356, 508  
*proc-target* (R742), 76, 77, 96, 162, **163**, 164, 165, 171, 319, 356, 499  
 procedure, **12**, **518**  
   characteristics of, 298  
   dummy, 298  
   elemental, 335  
   external, 297  
   internal, 297  
   intrinsic, 339–437  
   non-Fortran, 332  
   pure, 333  
 procedure designator, **22**, **518**  
 procedure interface, 299, **518**  
 procedure pointer, **22**, 306  
 procedure reference, **23**, 309  
 procedure references  
   generic, 489  
   resolving, 323  
*procedure-component-name*, 163  
*procedure-declaration-stmt* (R1211), 10, **306**, 307, 308, 493  
*procedure-designator* (R1221), 309, **309**, 325  
*procedure-entity-name*, 307  
*procedure-name*, 69, 70, 164, 165, 301, 307, 309, 310, 330  
*procedure-stmt* (R1206), 300, 301, **301**  
 processor, **1**, **518**  
 processor dependent, **3**, **518**  
 program, **12**, **518**  
*program* (R201), 9, **9**  
 program unit, **12**, **518**  
*program-name*, 289  
*program-stmt* (R1102), 9, 289, **289**  
*program-unit* (R202), 6, 9, **9**  
 PROTECTED attribute, **96**, 105

PROTECTED statement, 105, **105**  
*protected-stmt* (R547), 10, **105**  
 prototype, **518**  
 PUBLIC attribute, 57, 86  
 PUBLIC statement, 100  
 pure procedure, **333**, **518**

## Q

*query-spec* (R865), 202, **202**, 203  
*query-stmt* (R864), 11, **202**  
 QUOTE (DELIM value), 287

## R

*r* (R1005), 260, **260**, 261, 262  
 range, **186**  
 RANGE intrinsic, 47  
 rank, **20**, **21**, **518**  
*rbracket* (R471), 36, 61, 62, 80, **80**, 84, 89, 100, 103, 105, 123, 124  
 READ statement, **224**  
*read-stmt* (R910), 11, **224**, 225, 335, 508  
 READ= specifier, 253  
 reading, **207**  
 READWRITE= specifier, 253  
 REAL, 48  
 real and complex editing, 266  
 real model, 342  
 real part, **49**  
 real type, **47**, 47–49  
*real-literal-constant* (R413), 27, 48, **48**  
*real-part* (R418), 49, **49**  
 REC= specifier, 229  
 RECL= specifier, 221, 253  
 record, **207**, **518**  
 record file, **207**  
 record lengths, **208**  
 record number, **210**  
 RECURSIVE, 329  
 recursive input/output statement, **257**  
 reference, **518**  
*rel-op* (R713), 27, 135, **135**, 152  
 relational intrinsic operation, **138**  
 relational intrinsic operations, 152  
 relational intrinsic operator, **138**  
*rename* (R1111), 291, **291**, 292, 488  
*rep-char*, 52, **52**, 262, 279, 285  
 repeat specification., **260**  
 representable character, **52**  
 representation method, **47**  
 representation methods, **46**, **50**, **54**  
 resolving procedure references, 323  
   derived-type input/output, 243  
 restricted expression, **142**  
 result variable, **12**, **519**

*result-name*, 326, 327, 331, 493  
*result-token* (R334), 34, **34**  
 RETURN statement, 332  
*return-stmt* (R1240), 11, 15, 36, 186, 332, **332**  
 REWIND statement, 246  
*rewind-stmt* (R925), 11, **245**, 335  
*round-edit-desc* (R1018), 261, **262**  
**ROUND= specifier**, 221, 229, 253  
 rounding mode, **519**

## S

satisfied, **203**  
 SAVE attribute, 101, 105  
 SAVE statement, 105  
*save-stmt* (R548), 10, **105**, 493  
 saved, **98**  
*saved-entity* (R549), 105, **105**  
 scalar, **20**, **116**, **519**  
*scalar-xyz* (R103), 6, **6**  
 scale factor, **261**  
 scope, **487**, **519**  
 scoping unit, **12**, **519**  
 section subscript, **519**  
*section-subscript* (R620), 117, 120, **120**, 121  
 segment, **196**  
 SELECT CASE statement, 178  
 SELECT TYPE construct, 182  
**SELECT TYPE construct**, 492, 496  
*select-case-stmt* (R809), 178, **178**, 193  
*select-construct-name*, 182, 183  
*select-type-construct* (R821), 11, 182, **182**, 183  
*select-type-stmt* (R822), 182, **182**, 183, 193  
 SELECTED\_INT\_KIND intrinsic, 47  
 selector, **519**  
*selector* (R819), 180, 181, **181**, 182–184, 319, 496, 508  
 separate module procedure, **330**  
*separate-module-subprogram* (R1237), 10, 290, 330, **330**  
 sequence, **24**  
 sequence association, 318  
 SEQUENCE property, 59  
 SEQUENCE statement, **58**  
 sequence structure, **56**  
 sequence type, **58**  
*sequence-stmt* (R435), 57, **58**  
 sequential access, 209  
 sequential access input/output statement, **229**  
**SEQUENTIAL= specifier**, 253  
 shape, **21**, **519**  
 shape conformance, **141**  
*shared-term-do-construct* (R840), 186, **186**  
*sign* (R411), 47, **47**, 48, 267  
*sign-edit-desc* (R1016), 261, **261**

**SIGN= specifier**, 221, 229, 254  
*signed-digit-string* (R409), **47**, 49, 266, 267  
*signed-int-literal-constant* (R406), 47, **47**, 49, 50, 102, 261  
*signed-real-literal-constant* (R412), **48**, 50, 102  
*significand* (R414), 48, **49**  
 size, **21**, **519**  
 size of a common block, **112**  
 size of a storage sequence, **499**  
**SIZE= specifier**, 229, 254  
 source forms, **29**  
*source-expr* (R630), 124, **124**, 125, 126  
 special characters, **26**  
*special-character*, 25, **26**  
 specific interface, **301**  
 specific interface block, **302**  
 Specific names, **340**  
*specific-binding* (R454), 69, **69**  
 specification expression, **142**, **519**  
 specification function, **143**, **519**  
 specification inquiry, **142**  
*specification-expr* (R729), 43, 51, 84, 90, 142, **142**, 336  
*specification-part* (R204), 9, **10**, 69, 86, 99, 100, 143, 144, 191, 289, 290, 294, 301, 326, 329, 330, 334  
*specification-stmt* (R212), 10, **10**  
 specifications, 83–114  
 standard-conforming program, **2**, **519**  
 starting point, **116**  
*stat-variable* (R628), 124, **124**, 125, 126, 129, 198, 508  
 statement, **29**, **519**  
 statement entity, **487**, **519**  
 statement function, **298**, 333, **519**  
 Statement functions, 524  
 statement label, 28, **28**, **519**  
 statement order, 14  
 statements  
   accessibility, 99  
   ALLOCATABLE, 100  
   ALLOCATE, 124  
   arithmetic IF, 193  
   assignment, 156  
   ASSOCIATE, 180  
   ASYNCHRONOUS, 100  
   attribute specification, 99–114  
   BACKSPACE, 246  
   BIND, 100  
   CALL, 309  
   CASE, 178  
   CLASS IS, 182  
   CLOSE, 223  
   COMMON, 111–114



computed GO TO, 193  
 CONTAINS, 332  
 CONTINUE, 194  
 CYCLE, 188  
 DATA, 101  
 data transfer, 224  
 DEALLOCATE, 128  
 DIMENSION, 103  
 DO, 185  
 DO WHILE, 185  
 END, 15  
 ENDFILE, 246  
 ENTRY, 330  
 EQUIVALENCE, 108–111  
 EXTERNAL, 306  
 file positioning, 245  
 FLUSH, 247  
 FORALL, 168, 173  
 FORMAT, 259  
 formatted input/output, 226  
 FUNCTION, 326  
 GO TO, 193  
 IF, 177  
 IMPLICIT, 106  
 input/output, 207–257  
 INQUIRE, 248  
 INTENT, 103  
 INTRINSIC, 309  
 list-directed input/output, 227  
 NAMELIST, 108  
 namelist input/output, 227  
 NULLIFY, 128  
 OPEN, 217  
 OPTIONAL, 104  
 PARAMETER, 104  
 POINTER, 104  
 PRINT, 224  
 PRIVATE, 100  
 PROTECTED, 105  
 PUBLIC, 100  
 READ, 224  
 RETURN, 332  
 REWIND, 246  
 SAVE, 105  
 SELECT CASE, 178  
 SELECT TYPE, 182  
 STOP, 194  
 TARGET, 105  
 TYPE, 56  
 type declaration, 83–99  
 TYPE IS, 182  
 unformatted input/output, 226  
 VALUE, 105  
 VOLATILE, 105  
 WHERE, 166  
 WRITE, 224  
**STATUS= specifier**, 221, 224  
*stmt-function-stmt* (R1242), 10, 290, 294, 301, **333**, 493  
 STOP statement, 194  
*stop-code* (R856), 194, **194**, 195, 196  
*stop-stmt* (R855), 11, 36, 186, **194**, 335  
 storage associated, **500**  
 Storage association, **499**  
 storage association, 108–114, 499, **520**  
 storage sequence, 112, **499**, **520**  
 storage unit, **499**, **520**  
 stream access, 210  
 stream access input/output statement, **229**  
 stream file, **207**  
**STREAM= specifier**, 254  
 stride, **122**, **520**  
*stride* (R622), 120, **120**, 168, 169, 171, 173, 232  
 string, *see* character string  
 struct, **520**  
 structure, **19**, **56**, **520**  
 structure component, **117**, **520**  
 structure constructor, **76**, **520**  
*structure-component* (R614), 101, 115, 116, **118**, 124, 128  
*structure-constructor* (R460), **76**, 77, 102, 133, 334  
 subcomponent, **118**, **520**  
 submodule, **13**, **293**, **520**  
*submodule* (R1116), 9, **294**  
 submodule identifier, **293**  
*submodule-name*, 294  
*submodule-stmt* (R1117), 9, 293, 294, **294**  
 subobject, **520**  
 subobjects, **19**  
 subprogram, **520**  
 subroutine, **12**, **520**  
 subroutine reference, 322  
 subroutine subprogram, **329**, **520**  
*subroutine-name*, 301, 302, 329, 489  
*subroutine-stmt* (R1234), 9, 301, 302, 326, 329, **329**, 489, 493  
*subroutine-subprogram* (R1233), 9, 10, 290, **329**, 330  
 subscript, **520**  
*subscript* (R619), 7, 101, 120, **120**, 168, 169, 173, 232  
 subscript triplet, 122, **520**  
*subscript-triplet* (R621), 120, **120**, 121  
 substring, **116**, **520**  
*substring* (R609), 109, 115, **116**  
*substring-range* (R611), 88, 116, **116**, 118, 120, 232

*suffix* (R1231), 326, **327**, 330  
*sync-all-stmt* (R857), 11, **198**  
*sync-images-stmt* (R861), 11, **201**  
*sync-memory-stmt* (R866), 11, **204**  
*sync-stat* (R858), 198, **198**, 199, 201–204  
*sync-team-stmt* (R859), 11, **199**

## T

target, **22**, **520**  
 TARGET attribute, **98**, 105  
 TARGET statement, 105  
*target-decl* (R552), 105, **105**  
*target-stmt* (R551), 10, **105**, 493  
 team, **18**, **520**  
 team synchronization, **199**, **520**  
 terminal point, **211**  
 TKR compatible, **45**  
*token* (R335), 34, **35**  
 totally [storage] associated, **501**  
 transfer of control, 176, 193, 255, 256  
 transformational function, **520**  
 transformational functions, **339**  
 type, **18**, 41–81, **520**  
   base, 73  
   character, 50–54  
   complex, 49  
   concept, **41**  
   declared, 44  
   derived types, 78  
   dynamic, 44  
   expression, 139  
   extended, 73  
   extensible, 73  
   extension, 73  
   integer, 46  
   intrinsic types, 46–54  
   logical, 54  
   operation, 140  
   parent, 73  
   primary, 140  
   real, 47–49  
 type compatible, **44**, **521**  
 type conformance, 158  
 type declaration statement, **521**  
 type declaration statements, 83–99  
 type equality, 58  
 type incompatible, **44**  
 type parameter, **18**, 42, 47, **521**  
 type parameter inquiry, **119**  
 type parameter keyword, **22**  
 Type parameter order, **61**  
 type parameter order, **521**  
 type specifier, **43**  
   derived type, 44

TYPE, 44

TYPE type specifier, 44  
*type-attr-spec* (R432), 56, **56**, 57  
 type-bound procedure, **70**, **521**  
*type-bound-procedure-part* (R451), 56, 58, **68**,  
   71, 480  
*type-declaration-stmt* (R501), 10, 51, 83, **83**,  
   334, 493  
*type-guard-stmt* (R823), 182, **182**, 183  
*type-name*, 56, 57, 60, 69, 75, 521  
*type-param-attr-spec* (R438), 60, **60**  
*type-param-decl* (R437), 60, **60**  
*type-param-def-stmt* (R436), 56, 60, **60**  
*type-param-inquiry* (R616), 119, **119**, 133, 491  
*type-param-name*, 56, 60, 62, 119, 133, 134, 491,  
   493  
*type-param-spec* (R459), 75, **75**, 76  
*type-param-value* (R401), 42, **42**, 43, 51, 62, 75,  
   76, 124, 126  
*type-spec* (R402), 43, **43**, 44, 51, 52, 80, 81, 124–  
   126, 182

## U

ultimate component, **521**  
 ultimate components, **56**  
 unallocated, **127**  
 undefined, **23**, **521**  
 undefinition of variables, 503  
*underscore* (R303), 25, **26**  
 unformatted data transfer, 235  
 unformatted input/output statement, **226**  
 unformatted record, **208**  
**UNFORMATTED=** specifier, 254  
 Unicode, **219**  
 unit, **214**  
 unlimited polymorphic, **44**  
*unlimited-format-item* (R1004), 259, **260**, 263  
 unordered, **197**  
 unsaved, **98**  
 unsigned, **521**  
 unspecified storage unit, **499**, **521**  
*upper-bound* (R514), 90, **90**  
*upper-bound-expr* (R635), 124, **124**, 127, 163  
 use associated, **291**  
 Use association, **493**  
 use association, 493, **521**  
 USE statement, **291**  
*use-defined-operator* (R1115), 291, 292, **292**  
*use-name*, 99, 100, 291, 292, 488  
*use-stmt* (R1109), 10, **291**, 292, 493

## V

*v* (R1011), 240, 261, **261**, 274  
 value, **171**

VALUE attribute, **99**, 105  
 value separator, **278**  
 VALUE statement, 105  
*value-stmt* (R553), 11, **105**  
 variable, **521**  
*variable* (R601), 77, 101, 115, **115**, 157–160, 163, 168, 169, 171, 172, 181–183, 230, 310, 508  
*variable-name* (R602), 108, 109, 111, 112, 115, **115**, 116, 124, 128, 163, 493, 508  
 variables, **19**  
     definition & undefinition, 503  
 vector subscript, **122**, **521**  
*vector-subscript* (R623), 120, **120**, 121  
 void, **521**  
 VOLATILE attribute, **99**, 105  
 VOLATILE statement, 105  
*volatile-stmt* (R554), 11, **105**

## W

*w* (R1007), 260, 261, **261**, 265–269, 271–274, 279, 281, 282, 285, 287  
 wait operation, 223, 232, 244  
**WAIT statement**, 244  
*wait-spec* (R922), 244, **244**  
*wait-stmt* (R921), 11, **244**, 335  
 WHERE construct, 166  
 WHERE statement, 166  
*where-assignment-stmt* (R747), 166, **166**, 168, 172, 512  
*where-body-construct* (R746), 166, **166**, 167, 168  
*where-construct* (R744), 11, 166, **166**, 169, 172  
*where-construct-name*, 166  
*where-construct-stmt* (R745), 166, **166**, 167, 172, 193  
*where-stmt* (R743), 11, 166, **166**, 169, 172  
 whole array, **119**, **521**  
 WRITE statement, **224**  
*write-stmt* (R911), 11, **224**, 225, 335, 508  
**WRITE= specifier**, 254  
 writing, **207**

## X

*xyz-list* (R101), **6**  
*xyz-name* (R102), **6**

## Z

zero-size array, 21, 90, 102  
 ZZZUTI1, 17  
 ZZZUTI10, 98  
 ZZZUTI11, 88  
 ZZZUTI12, 89  
 ZZZUTI13, 89  
 ZZZUTI14, 44

ZZZUTI15, 45  
 ZZZUTI16, 45  
 ZZZUTI17, 45  
 ZZZUTI18, 45  
 ZZZUTI19, 46  
 ZZZUTI2, 18  
 ZZZUTI20, 117  
 ZZZUTI21, 123  
 ZZZUTI22, 125  
 ZZZUTI23, 157  
 ZZZUTI24, 157  
 ZZZUTI25, 188  
 ZZZUTI26, 194  
 ZZZUTI27, 194  
 ZZZUTI28, 194  
 ZZZUTI29, 195  
 ZZZUTI3, 16  
 ZZZUTI30, 195  
 ZZZUTI31, 195  
 ZZZUTI32, 195  
 ZZZUTI33, 196  
 ZZZUTI34, 197  
 ZZZUTI35, 200  
 ZZZUTI36, 201  
 ZZZUTI37, 204  
 ZZZUTI38, 206  
 ZZZUTI39, 216  
 ZZZUTI4, 16  
 ZZZUTI40, 222  
 ZZZUTI41, 222  
 ZZZUTI42, 223  
 ZZZUTI43, 223  
 ZZZUTI44, 251  
 ZZZUTI45, 266  
 ZZZUTI46, 266  
 ZZZUTI47, 266  
 ZZZUTI48, 282  
 ZZZUTI49, 300  
 ZZZUTI5, 17  
 ZZZUTI50, 313  
 ZZZUTI5003, 517  
 ZZZUTI51, 313  
 ZZZUTI52, 314  
 ZZZUTI53, 314  
 ZZZUTI54, 317  
 ZZZUTI55, 316  
 ZZZUTI56, 339  
 ZZZUTI57, 339  
 ZZZUTI58, 340  
 ZZZUTI59, 348  
 ZZZUTI6, 17  
 ZZZUTI60, 361  
 ZZZUTI61, 361  
 ZZZUTI62, 359

ZZZUTI63, 382  
ZZZUTI64, 376  
ZZZUTI65, 395  
ZZZUTI66, 400  
ZZZUTI67, 405  
ZZZUTI68, 405  
ZZZUTI69, 414  
ZZZUTI7, 21  
ZZZUTI70, 418  
ZZZUTI71, 427  
ZZZUTI72, 427  
ZZZUTI73, 439  
ZZZUTI74, 472  
ZZZUTI75, 497  
ZZZUTI76, 497  
ZZZUTI77, 500  
ZZZUTI78, 515  
ZZZUTI79, 317  
ZZZUTI8, 97  
ZZZUTI9, 97

## **Index**