FUJITSU

FUJITSU Software
Technical Computing Suite V2.0

Fortran Language Reference

J2UL-1864-02ENZ0(00)
November 2015

Preface

Purpose of This Manual

This manual is intended as a reference to the Fortran 2003 language for programmers with experience in Fortran.
For information on creating programs using the Fujitsu Fortran System, see "Fortran User's Guide".
Fujitsu Fortran supports the Fortran 2003 standard and extensions.
Fujitsu Fortran conforms to

ISO/IEC 1539-1:2004, Information technology - Programming languages - Fortran

Intended Readers

This manual is intended for the users who compile a Fortran program.

Organization of This Manual

This manual is organized as follows:

- Chapter 1 Elements of Fortran
  This chapter takes an elemental, building-block approach, starting from Fortran's smallest elements, its character set, and proceeding through source form, data, expressions, input/output, statements, executable constructs, and procedures, and ending with program units.
- Chapter 2 Alphabetical Reference
  This chapter gives detailed syntax and constraints for Fortran statements, constructs, intrinsic procedures, and service routines.
- Appendix A Intrinsic Procedures
  This chapter is a table containing brief descriptions and specific names of intrinsic procedures included with this system.
- Appendix B Service Routines
  This chapter is a table containing brief descriptions and specific names of service routines included with this system.
- Appendix C Extended Features
  This chapter is a table supporting extensions from Fortran 2003.
- Appendix D Glossary
  This chapter defines various technical terms used in this manual.
- Appendix E ASCII Character Set
  This chapter details the 128 characters of the ASCII set.
- Appendix F Fortran 2003 Additional Specifications
  This chapter is a table support and description for Fortran 2003 specifications.
- Appendix G Limitations
  This chapter is a table containing limitations for Fortran 2003 specifications.
- Appendix H Support Fortran 2008 Specifications
  This chapter is a table support for Fortran 2008 specifications.

Notation Used in This Manual

The following conventions are used throughout the manual:

blue text	indicates an extension to the Fortran 2003 standard.
"blue text"	Hyperlink
`Program`	indicates Fortran source code.

In syntax descriptions,

KEYWORD	descriptions are to be entered exactly as they appear.
Italics	indicate text to be replaced by you.

[ ]	enclose optional items.
...	following an item indicates that more items of the same form may appear.
abc-list	same as *abc* [,*abc*] ... .
&	continues a syntax rule.

Export Controls

Exportation/release of this document may require necessary procedures in accordance with the regulations of your resident country and/or US export control laws.

Trademarks

OpenMP is a trademark of OpenMP Architecture Review Board.
All other trademarks and product names are the property of their respective owners.
The trademark notice symbol (TM,(R)) is not necessarily added in the system name and the product name, etc. published in this material.

Date of Publication and Version

Version	Manual code
November 2015, 2nd Version	J2UL-1864-02ENZ0(00)
February 2015, Version 1.1	J2UL-1864-01ENZ0(01)
October 2014, 1st Version	J2UL-1864-01ENZ0(00)

Copyright

Update History

Changes	Location	Version
Add the following Fortran 2008 specifications: - BLOCK construct, END BLOCK statement - COARRAY specifications - ERROR STOP statement - Implied shape array - Double colon after PROCEDURE in interface block - Scalar default CHARACTER initialization expression and scalar INTEGER initialization expression in STOP statement	At the whole	2nd Version
The descriptions are modified.	Derived Type Definition Structure Component Elemental Procedure Declaration Dummy Data Objects ALLOCATE Statement CHMOD Service Function	

Changes	Location	Version
	Simply CONTIGUOUS CSHIFT Intrinsic Function SAVE Statement SELECT TYPE Construct TYPE Statement WHERE Statement Table A.2 Mathematical Intrinsic Functions	
Syntax rule is added.	Preface	
Descriptions are modified.	Structure Component	
Descriptions are added.	Derived Type Specifier	
Syntax rule is modified.	Procedure Interface Block	
Descriptions are modified.	Statement Labels	
List is modified.	Appendix F Appendix G	
Fixed the error in writing.	At the whole	
SIZE is modified.	RANDOM_SEED Intrinsic Subroutine	Version 1.1

Refer to the following for the update history of Technical Computing Suite V1.0.

Edition	Date	Revised Location	Description
1	2012-06-29	At the whole	- New specifications of the Fortran 2003 standard are added. - "CHARACTER scalar" is changed to "Default CHARACTER scalar".
		FGETC Service Function FPUTC Service Function FSEEK Service Function FSEEKO64 Service Function GETC Service Function GETFD Service Function PUTC Service Function	- Result is fixed.
		F Fortran 2003 additional specifications	- List is modified.
		G Limitations	- New.
2	2012-12-10	Binary, Octal, and Hexadecimal Constants	- The description of intrinsic function is added.
		Allocatable Array Association Status	- The conditions concerning the ALLOCATE statement are added.
		Assumed-Size Array Specification Expression Pure Procedures IEEE Exception ACOSH Intrinsic Function ASINH Intrinsic Function ATANH Intrinsic Function	- The descriptions are added.

Edition	Date	Revised Location	Description
		BGE Intrinsic Function BGT Intrinsic Function BLE Intrinsic Function BLT Intrinsic Function DSHIFTL Intrinsic Function DSHIFTR Intrinsic Function END ASSOCIATE Statement HYPOT Intrinsic Function LEADZ Intrinsic Function MASKL Intrinsic Function MASKR Intrinsic Function MERGE_BITS Intrinsic Function POPCNT Intrinsic Function POPPAR Intrinsic Function SHIFTA Intrinsic Function SHIFTL Intrinsic Function SHIFTR Intrinsic Function STORAGE_SIZE Intrinsic Function TRAILZ Intrinsic Function CLASS DEFAULT Statement CLASS IS Statement TYPE IS Statement FORALL Construct	
		Type-Bound Procedure	- The descriptions concerning FINAL are added.
		Expressions	- The descriptions concerning the operator are added.
		Initialization Expression	- The descriptions of intrinsic function are added.
		Executable Statements	- END ASSOCIATE is added.
		Dummy Data Objects	- Sentences of a blue character are changed to the black.
		Procedure Interface Block	- The description concerning procedure-name-list is added.
		Defined Assignment	- The descriptions concerning subroutine are added.
		Standard Intrinsic Module	- Intrinsic module function and intrinsic module subroutine are added.
		Scope	- The descriptions concerning construct are added.
		Alphabetical Reference	- "Arguments" is changed to "Argument(s)".
		ACOS Intrinsic Function ASIN Intrinsic Function COSH Intrinsic Function IAND Intrinsic Function IEOR Intrinsic Function IOR Intrinsic Function SINH Intrinsic Function TAN Intrinsic Function TANH Intrinsic Function	- The specifications are modified.

Edition	Date	Revised Location	Description
		ACOSD Intrinsic Function ACOSQ Intrinsic Function AIMAG Intrinsic Function ANINT Intrinsic Function GETC Service Function GET_COMMAND Intrinsic Subroutine GET_COMMAND_ARGUMENT Intrinsic Subroutine GET_ENVIRONMENT_VARIABLE Intrinsic Subroutine	- INTEGER, CHARACTER, REAL, COMPLEX and LOGICAL are changed from the small letter to the capital letter.
		ASSOCIATE Construct Form	- The description of "association" is added.
		Associate Name Attributes ASYNCHRONOUS Statement	- Representations of sentences are modified.
		ATAN Intrinsic Function	- COMPLEX type argument is added. - ATAN(Y,X) is added.
		DO Construct	- Remarks are fixed.
		IEEE_GET_FLAG Intrinsic Module Subroutine	- Argument(s) is fixed.
		IEEE_SUPPORT_DIVIDE Intrinsic Module Function SPACING Intrinsic Function	- Remarks are fixed.
		MINVAL Intrinsic Function	- The explanation of result value is fixed.
		MVBITS Intrinsic Subroutine	- FROMPOS and TOPOS are modified.
		NULL Intrinsic Function	- MOLD is modified.
		REAL Intrinsic Function	- Data type of result is fixed.
		SELECT TYPE Construct	- The description concerning derived type specifier is added.
		A Intrinsic Procedures C Extended Features	- New intrinsic functions are added.
		D Glossary	- "initialization expression" is added. - "restricted expression" is added.
		F Fortran 2003 additional specifications G Limitations	- List is modified.
		Index	- Index is modified.
3	2013-04-18	INTEGER Literal Constants	- The explanation of -2147483648 is added.
4	2013-11-11	At the whole	- Errata are fixed.
		Manual Organization	- H Support Fortran 2008 specifications is added.
		Attributes	- CONTIGUOUS attribute is added.
		Derived Type Definition Optional Arguments Dummy Data Object Procedure Interfaces Scope C_LOC Intrinsic module function DO Construct DO Statement	- The descriptions are added.

Edition	Date	Revised Location	Description
		ISO_C_BINDING Intrinsic Module NULL Intrinsic Function Data Pointer Assignment Type Declaration Statement	
		Nonexecutable Statement	- CONTIGUOUS is added.
		Argument Intent ALLOCATE Statement Associate Name Attribute FORALL Construct FORALL Construct Statement FORALL Statement INTENT Statement SELECT TYPE Construct	- The descriptions are modified.
		FORALL header Determination of the values for index variables FORALL mask expression C_SIZEOF Intrinsic module function COMPILER_OPTIONS Intrinsic Module Function COMPILER_VERSION Intrinsic Module Function CONTIGUOUS Statement Simply CONTIGUOUS IS_CONTIGUOUS Intrinsic Function	- New.
		DEALLOCATE Statement	- Remarks are modified.
		ISO_FORTRAN_ENV Intrinsic Module	- Rename - List is modified.
		A Intrinsic Procedures C Extended Features	- List is modified
		D Glossary	- The descriptions are modified.
		H Support Fortran 2008 specifications	- New.
5	2014-10-01	At the whole	- Reworked the format all over with the version upgrade of Technical Computing Suite V2.0.
		Export Controls	- New.
		ABORT Service Subroutine	- The description is modified.
		Modules AUTOMATIC Statement	- The descriptions are added.
		DATE_AND_TIME Intrinsic Subroutine	- Example is modified.
		IEEE_SCALB Intrinsic Module Function	- Errata are fixed.
		LRSHFT Intrinsic Function LSHIFT Intrinsic Function RSHIFT Intrinsic Function	- Argument(s) are fixed.
		Appendix C Extended Features Appendix H Support Fortran 2008 specifications	- [Attribute] is modified.
		Index	- Index is modified.

Contents

Chapter 1 Elements of Fortran

1.1 Character Set

The Fortran character set consists of

- letters:

```
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
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
```

- digits:

```
0 1 2 3 4 5 6 7 8 9
```

- special characters:

```
<blank> = + - * / \ ( ) [ ] { } , . : ; ! " % & ~ < > ? ' ` ^ | $ # @
```

- and the underscore character '_'.

Lower case letters are equivalent to corresponding upper-case letters except in CHARACTER literals.

The underscore character can be used as a non-leading significant character in a name.

Special characters are used as operators, as separators or delimiters, or for grouping.

1.2 Names

Names are used in Fortran to refer to various entities such as variables and program units. A name starts with a letter or a '$', can be up to 240 characters in length and consists entirely of letters, digits, underscores, and '$'.

Examples of legal Fortran names are:

```
aAaAa        apples_and_oranges        r2d2
rose            ROSE            Rose
```

The three representations for the names on the line immediately above are equivalent.

The following names are illegal:

```
_start_with_underscore
2start_with_a_digit
illegal_@_character
```

1.3 Statement Labels

Fortran statements can have labels consisting of one to five digits, at least one of which is non-zero. The same statement label shall not be given to more than one statement in a scoping unit. Leading zeros are not significant in distinguishing statement labels.

A statement label provides a means of referring to an individual statement. Only branch target statements, FORMAT statements, and DO terminations shall be referred to by the use of statement labels. A branch target statement is an executable statement except ELSE statements, ELSE IF statements, CASE statements, END FORALL statements, ELSEWHERE statements, END WHERE statements, and type guard statements.

The following labels are valid:

```
123
50000
10
010
```

The last two labels are equivalent. The same statement label shall not be given to more than one statement in a scoping unit.

1.4 Source Form

Fortran offers two source forms: free and fixed.

A character context means characters within a character literal constant (see "1.5.4.5 CHARACTER Literal Constants") or with in a character string edit descriptor (see "1.8.1.4 Character String Edit Descriptors").

1.4.1 Free Source Form

In free source form, there are no restrictions on where a statement can appear on a line. A line can be up to 255 characters long. Except in character contexts, blanks are ignored.

The '!' character begins a comment except when it appears in a character context. The comment extends to the end of the line.

The ';' character can be used to separate statements on a line. If it appears in a character context or in a comment, the ';' character is not interpreted as a statement separator.

The '&' character as the last non-comment, non-blank character on a line indicates the line is to be continued on the next non-comment line. If a name, constant, keyword, or label is split across the end of a line, the first non-blank character on the next non-comment line shall be the '&' character followed by successive characters of the name, constant, keyword, or label. If a character context is to be continued, the '&' character ending the line cannot be followed by a trailing comment.

A free source form statement can have unlimited continuation lines.

Comment lines cannot be continued, but a continuation line can contain a trailing comment. A line cannot contain only an '&' character or contain an '&' character as the only character before a comment.

Beginning a line with a ' " ', an ' * ', or a ' + ' column 1 is a comment line. The ' - ' character as the last non-comment, non-blank character on a line indicates the line is to be continued on the next line column 1.

1.4.2 Fixed Source Form (obsolescent feature)

In fixed source form, there are restrictions on where statements and labels can appear on a line. Except in character contexts, blanks are ignored.

Except within a comment:

- Columns 1 through 5 are reserved for statement labels. If a label does not present, columns 1 though 5 shall be blank. Labels can contain blanks. Columns 1 though 5 of any continuation lines shall be blank.
- Column 6 is used only to indicate a continuation line. If column 6 contains a blank or zero, column 7 begins a new statement. If column 6 contains any other character, columns 7 through 72 are a continuation of the previous non-comment line. There can be unlimited continuation lines. Continuation lines shall not be labeled.
- Columns 7 through 72 are used for Fortran statements.
- Columns after 72 are ignored.

The ' ! ' character begins a comment except when it appears in a character context or column 6. The comment extends to the end of the line. Additionally, fixed source form comments are formed by beginning a line with a ' C ' or a ' * ' in column 1. Comment lines shall not be continued, but a continuation line can contain a trailing comment. An END statement shall not be continued.

The ' ; ' character can be used to separate statements on a line. If it appears in a character context, in a comment, or column 6, the ' ; ' character is not interpreted as a statement separator.

1.5 Data

A data has a data type and may have any of the set of values specified for its data type. A data type is characterized by a set of values, together with a way to denote these values and a collection of operations that interpret and manipulate the values. Constants, variables, and subobjects of a constant are data objects. Data objects, the result of the evaluation of an expression, and function results are data entity. A data entity has a data type and has, or may have, a data value except an undefined variable. Every data entity has a rank and it's thus either a scalar or an array. A named data object may have attributes declared in type declaration statement or individually specification statements.

A type is specified by type specifiers. There are intrinsic type specifier (see "2.469 Type Declaration Statement") and derived type specifier (see "1.5.11.8 Derived Type Specifier").

1.5.1 Intrinsic Data Types

The five intrinsic data types are INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER. INTEGER, REAL, and COMPLEX types are numeric types. LOGICAL and CHARACTER types are nonnumeric types.

1.5.2 Kind

An intrinsic data type has one or more kinds. In this system for the CHARACTER, INTEGER, REAL, and LOGICAL data types, the kind type parameter (a number used to refer to a kind) corresponds to the number of bytes used to represent each respective kind. For the COMPLEX data type, the kind type parameter is the number of bytes used to represent the real or the imaginary part.

If the kind type parameter is not specified, the default kind value is determined implicitly. For INTEGER, REAL, COMPLEX and LOGICAL types, the default kind value is 4, and for CHARACTER type, it is 1. These each type is default INTEGER, default REAL, default COMPLEX, default LOGICAL, and default CHARACTER.

Three intrinsic inquiry functions, SELECTED_CHAR_KIND (see "2.410 SELECTED_CHAR_KIND Intrinsic Function"), SELECTED_INT_KIND (see "2.411 SELECTED_INT_KIND Intrinsic Function") and SELECTED_REAL_KIND (see "2.412 SELECTED_REAL_KIND Intrinsic Function"), are provided. Each returns a kind type parameter based on the required range and precision of a data object in a way that is portable to other Fortran systems. The kinds available in this system are summarized in the following table:

Table 1.1 Intrinsic Data Types

Type	Kind Type Parameter	Type Name	Notes
INTEGER	1	One-byte INTEGER	Range: -128 to 127
	2	Two-byte INTEGER	Range: -32,768 to 32,767
	4 (default INTEGER)	Four-byte INTEGER	Range: -2,147,483,648 to 2,147,483,647
	8	Eight-byte INTEGER	Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
REAL	4 (default REAL)	Single precision REAL	Range: $1.18*10^{-38}$ to $3.40*10^{38}$ Precision: 7-8 decimal digits
	8	Double precision REAL	Range: $2.23*10^{-308}$ to $1.79*10^{308}$ Precision: 15-16 decimal digits
	16	Quadruple precision REAL	Range: 10^{-4931} to 10^{4932} Precision: approximately 33 decimal digits
COMPLEX	4 (default COMPLEX)	Single precision COMPLEX	Range: $1.18*10^{-38}$ to $3.40*10^{38}$ Precision: 7-8 decimal digits
	8	Double precision COMPLEX	Range: $2.23*10^{-308}$ to $1.79*10^{308}$ Precision: 15-16 decimal digits
	16	Quadruple precision COMPLEX	Range: 10^{-4931} to 10^{4932} Precision: approximately 33 decimal digits
LOGICAL	1	One-byte LOGICAL	Values: .TRUE. and .FALSE.
	2	Two-byte LOGICAL	Values: .TRUE. and .FALSE.
	4 (default LOGICAL)	Four-byte LOGICAL	Values: .TRUE. and .FALSE.

Type	Kind Type Parameter	Type Name	Notes
	8	Eight-byte LOGICAL	Values: .TRUE. and .FALSE.
CHARACTER	1 (default CHARACTER)	ASCII CHARACTER	ASCII character set
	4	ISO 10646 CHARACTER	ISO 10646 character set

1.5.3 Character Length

The number of characters in a CHARACTER data object is indicated by its length type parameter. For example, the CHARACTER literal constant "`Character data`" has a length of fourteen.

1.5.4 Literal Constants

A literal datum, also known as a literal, literal constant, or immediate constant, is specified as follows for each of the Fortran data types. The syntax of a literal constant determines its intrinsic type.

1.5.4.1 INTEGER Literal Constants

An INTEGER literal constant consists of one or more digits preceded by an optional sign ('+' or '-') and followed by an optional underscore '_' and kind type parameter. If the optional underscore '_' and kind type parameter are not present, the INTEGER literal is of type default INTEGER (four-byte INTEGER in this system).

Examples of valid INTEGER literals are

```
34     -256     345_4     +78_mykind
```

`34` and `-256` are of type default INTEGER (four-byte INTEGER in this system). `345_4` is of type four-byte INTEGER. In the last example, `mykind` shall have been previously declared as a scalar INTEGER named constant with the value of an INTEGER kind type parameter (1, 2, 4, or 8 in this system).

An INTEGER literal constant without kind type parameter shall not be -2147483648.

1.5.4.2 REAL Literal Constants

A REAL literal consists of one or more digits containing a decimal point '.' (the decimal point can appear before, within, or after the digits), optionally preceded by a sign ('+' or '-'), and optionally followed by an exponent letter and exponent, optionally followed by an underscore '_' and kind type parameter. If an exponent letter is present the decimal point '.' is optional. The exponent letter is 'E' for single precision, 'D' for double precision, and 'Q' for quadruple precision. If the optional underscore '_' and kind type parameter are present after exponent letter and exponent, the exponent letter shall be 'E'. If the optional underscore '_' and kind type parameter are not present, the REAL literal is type default REAL (single precision REAL in this system).

Examples of valid REAL literals are

```
-3.45     .0001     34.E-4     1.4_8
```

`-3.45` and `.0001` are of type default REAL (single precision REAL in this system). `34.E-4` is of type single precision REAL. The last example is of type double precision REAL.

1.5.4.3 COMPLEX Literal Constants

A COMPLEX literal constant is formed by enclosing in parentheses a comma-separated pair of REAL, INTEGER literal constants or named constants. The named constant must be of type REAL or INTEGER. The first of the REAL, INTEGER literal constants or named constants represents the real part of the COMPLEX literal constant; the second represents the imaginary part. The kind type parameter of a COMPLEX literal constant is 16 if either the real or the imaginary part or both are quadruple precision REAL, 8 if either the real or the imaginary part or both are double precision REAL, otherwise the kind type parameter is 4 (default COMPLEX).

Examples of valid COMPLEX literal constants are

```
(3.4,-5.45)   (-1,-3)   (3.4,-5)   (-3.d13,6._8)   (p,q)
```

The first three examples are of type default COMPLEX (single precision COMPLEX in this system, where four bytes are used to represent each part, real or imaginary, of the COMPLEX literal constant). The fourth and fifth examples are of type double precision COMPLEX, and use eight bytes for each part. The named constants p and q are of type double precision REAL.

1.5.4.4 LOGICAL Literal Constants

A LOGICAL literal is either .TRUE. or .FALSE., optionally followed by an underscore '_' and a kind type parameter. If the optional underscore '_' and kind type parameter are not present, the LOGICAL literal is of type default LOGICAL (four-byte LOGICAL in this system).

Examples of valid LOGICAL literals are

```
.false.     .true.     .false._4     .true._mykind
```

The first two examples are of type default LOGICAL (four-byte LOGICAL in this system). `.false._4` is of type four-byte LOGICAL. In the last example, `mykind` shall have been previously declared as a scalar INTEGER named constant with the value of a LOGICAL kind type parameter (1, 2, 4, or 8 in this system).

1.5.4.5 CHARACTER Literal Constants

A CHARACTER literal consists of a string of characters enclosed in matching apostrophes ' ' ' or quotation marks ' " ', optionally preceded by a kind type parameter and an underscore '_'.

If a quotation mark is needed within a CHARACTER string enclosed in quotation marks, double the quotation mark inside the string. The doubled quotation mark is then counted as a single quotation mark. Similarly, if an apostrophe is needed within a CHARACTER string enclosed in apostrophes, double the apostrophe inside the string. The double apostrophe is then counted as a single apostrophe.

Examples of valid CHARACTER literals are

```
"Hello world"
'don''t give up'
1_"Fortran"
mykind_'FUJITSU LIMITED'
''
""
```

`mykind` shall have been previously declared as a scalar INTEGER named constant with the value 1 or 4 to indicate the kind. The last two examples, which have no intervening characters between the quotes or apostrophes, are zero-length CHARACTER literals.

A Hollerith constant is a string that consists of an unsigned integer constant (greater than zero) followed by the letter 'H' or 'h' and then followed by any character string of length value of specified integer constant. A Hollerith constant is equivalent to a character constant unless otherwise noted.

A Hollerith constant shall not appear in:

- An operand in a STOP or PAUSE statement
- A format identifier in a PRINT or READ statement that does not specify a UNIT= specifier.

An escape sequence shall not appear in Hollerith constants.

In free source form, if the character '&' or '-' appears in a Hollerith constant as the last nonblank character on the line, the character '&' or '-' indicates that the statement continues on the next non-comment line.

Care is required when ISO_10646 character type character literal constants are used in a fixed format, as the number of digits may be exceeded.

1.5.4.5.1 Escape Sequences

In a character context, the following backslash escapes are recognized. The escape sequence treated as one indicated character, which is in a character literal constant or apostrophes contains a character constant edit descriptor, that contains two characters, a backslash followed by one character. An escape sequence shall not appear in Hollerith constant and H edit descriptor.

Escape Sequence	Character
\0	Null
\b	Backspace
\t	Horizontal tabulation
\n	Line feed
\f	Form feed
\"	Quotation
\'	Apostrophe
\\	Backslash
x	*x*, where *x* is any other character

The back slash character can be controlled to treat, escape sequence or ordinary character. See "Fortran User's Guide" for compiler option.

For example,

```
i = len('abc\n')        ! i is assigned the value 4
```

1.5.4.6 Binary, Octal, and Hexadecimal Constants

A binary, octal, or hexadecimal constant can appear in the followings.

- A right-hand side of an assignment expression
- A DATA statement
- A PARAMETER statement
- An initialization in a type declaration statement
- An argument A in an intrinsic function DBLE (see "2.109 DBLE Intrinsic Function"), REAL (see "2.390 REAL Intrinsic Function") or INT (see "2.273 INT Intrinsic Function")
- An argument X or Y in an intrinsic function CMPLX (see "2.72 CMPLX Intrinsic Function")
- An argument I or J in an intrinsic function BGE (see "2.42 BGE Intrinsic Function"), BGT (see "2.43 BGT Intrinsic Function"), BLE (see "2.49 BLE Intrinsic Function"), BLT (see "2.52 BLT Intrinsic Function"), DSHIFTL (see "2.120 DSHIFTL Intrinsic Function"), DSHIFTR (see "2.121 DSHIFTR Intrinsic Function"), IAND (see "2.215 IAND Intrinsic Function"), IEOR (see "2.260 IEOR Intrinsic Function"), IOR (see "2.279 IOR Intrinsic Function") or MERGE_BITS (see "2.333 MERGE_BITS Intrinsic Function").

If a binary, octal, or hexadecimal constant appears in a right-hand side of an assignment expression, a DATA statement, in a PARAMETER statement, and an initialization in a type declaration statement, a type of assignment variable shall be corresponding to an intrinsic. Also, a binary, octal, or hexadecimal constant will be assigned in variable area directly. If it is shorter than the size of corresponding variable, the space to the left the constant is padded with zeroes. If it exceeds the size of corresponding variable, the high-order excess is truncated.

If an actual argument of it is shorter than the size of corresponding type of a result of an intrinsic function, the space to the left the constant is padded with zeroes. If it exceeds the size of corresponding type of a result of an intrinsic function, the high-order excess is truncated.

1.5.4.6.1 Binary Constants

A binary constant is following syntax:

```
B'digits'       or
B"digits"
```

Where:

The *digits* is a string of zero or more binary digits. A binary digit is the digit 0 or 1.

For example,

```
data i /b'0001'/       ! i is initialized the value 1
parameter (j=b'0010')  ! j is assigned the value 2
```

1.5.4.6.2 Octal Constants

An octal constant is following syntax:

```
O'digits'        or
O"digits"        or
'digits'O        or
"digits"O
```

Where:

The *digits* is a string of zero or more octal digits. An octal digit is a digit in the range 0 to 7.

For example,

```
data i /o'001'/       ! i is initialized the value 1
data j /o"010"/       ! j is initialized the value 8
parameter (k='015'o)  ! k is assigned the value 13
```

1.5.4.6.3 Hexadecimal Constants

A hexadecimal constant is following syntax:

```
Z'digits'     or
Z"digits"     or
X'digits'     or
X"digits"     or
'digits'X     or
"digits"X
```

Where:

The *digits* is a string of zero or more hexadecimal digits. A hexadecimal digit is a digit or a letter in the range A to F. The letter in a hexadecimal digit can be small letter.

For example,

```
data i /z'01'/        ! i is initialized the value 1
data j /z"10"/        ! j is initialized the value 16
parameter (k='1a'x)   ! k is assigned the value 26
```

1.5.5 Named Data

A named data object, such as a variable, named constant, or function result, is given the properties of an intrinsic or user-defined data type, either implicitly (based on the first letter of the name) or through a type declaration statement. Additional information about a named data object, known as the data object's attributes, can also be specified, either in a type declaration statement or in separate statements specific to the attributes that apply.

1.5.5.1 Implicit Typing

In the absence of a type declaration statement, a named data object's type is determined by the first letter of its name. The letters I through N begin data objects of type default INTEGER and the other letters and a '$' begin data objects of type default REAL. These implicit typing rules can be customized or disabled using the IMPLICIT statement. IMPLICIT NONE can be used to disable all implicit typing for a scoping unit.

1.5.5.2 Explicitly Type Declaration

A type declaration statement specifies the type, type parameters, and attributes of a named data object or function. A type declaration statement is available for each intrinsic type, INTEGER, REAL, COMPLEX, LOGICAL, or CHARACTER, as well as for derived types (see "1.5.11 Derived Types").

1.5.5.3 Attributes

Besides type and type parameters, a data object or function can have one or more of the following attributes, which can be specified in a type declaration statement or in a separate statement particular to the attribute. A named data object shall not be explicitly specified to have a particular attribute more than once in a scoping unit.

- ALLOCATABLE - the data object is an allocatable variable, but is to have memory allocated for it as specified during execution of the program (see "2.15 ALLOCATABLE Statement").
- ASYNCHRONOUS - asynchronous input/output is used to handle variables (see "2.30 ASYNCHRONOUS Statement").
- AUTOMATIC - the data object to be on the stack (see "2.40 AUTOMATIC Statement").
- BIND - the variable or common block is a C language variable with an external association and is interoperable (see "2.45 BIND Statement").
- CHANGEENTRY - the name is that of an external procedure except Fortran program (see "2.60 CHANGEENTRY Statement").
- CODIMENSION - the data object is a coarray (see "2.73 CODIMENSION Statement").
- CONTIGUOUS - the array pointer or assumed shape array is CONTIGUOUS (see "2.82 CONTIGUOUS Statement").
- DIMENSION - the data object is an array (see "2.113 DIMENSION Statement").
- EXTERNAL - the name is that of an external procedure, a dummy procedure, or procedure pointer, and it can be used as an actual argument (see "2.169 EXTERNAL Statement").
- INTENT - the dummy argument shall neither be defined nor become undefined in the procedure (INTENT(IN)), shall be defined before a reference and any actual argument that becomes associated with such a dummy argument shall be definable (INTENT(OUT)), or is intended for use both to receive data from and to return data to the invoking scoping unit (INTENT(INOUT)) (see "2.275 INTENT Statement").
- INTRINSIC - the name is that of an intrinsic procedure (see "2.277 INTRINSIC Statement").
- OPTIONAL - the dummy argument need not have a corresponding actual argument in a reference to the procedure in which the dummy argument appears (see "2.355 OPTIONAL Statement").
- PARAMETER - the data object is a named constant (see "2.358 PARAMETER Statement").
- POINTER - the data object is a pointer (see "2.361 POINTER Statement").
- PRIVATE - the named data object, a procedure or a type in a MODULE program unit is accessible only in the current module (see "2.370 PRIVATE Statement").
- PROTECTED - the named variable is specified in a MODULE program unit, the part in the useful range by use association where the variable can be modified is limited (see "2.377 PROTECTED Statement").
- PUBLIC - the named data object, a procedure or a type in a MODULE program unit is accessible in a program unit that uses that module (see "2.378 PUBLIC Statement").
- SAVE - the data object retains its value, definition status, association status, and allocation status after a RETURN or END statement (see "2.404 SAVE Statement").
- STATIC - the data object to be in static memory (see "2.442 STATIC Statement").
- TARGET - the data object that is to be associated with a pointer (see "2.458 TARGET Statement").
- VALUE - the dummy argument to be associated with a definable temporary area whose initial value is that of the actual argument. Neither the value nor the changes in the definition status for the dummy argument influence the actual argument (see "2.481 VALUE Statement").
- VOLATILE - the data object is prevented optimization (see "2.483 VOLATILE Statement").

1.5.6 Scalar

A Scalar is a data entity that can be represented by a single value of the data type and that is not an array. The rank of a scalar is zero.

1.5.6.1 Scalar Pointer

A scalar pointer can be allocated as needed using an ALLOCATE statement, and deallocated with a DEALLOCATE statement. Alternatively, a scalar pointer can be associated with a target using a pointer assignment statement. The rank of a scalar pointer is 0.

1.5.6.2 Scalar Allocatable Variable

A scalar allocatable variable can be allocated as needed using an ALLOCATE statement, and deallocated with a DEALLOCATE statement. The rank of a scalar allocatable variable is 0. The association status of a scalar allocatable variable is the same as that of allocatable arrays (see "1.5.9.1.1 Allocatable Array Association Status").

1.5.7 Substrings

A character string is a sequence of characters in a CHARACTER data object. The characters in the string are numbered from left to right starting with one. A contiguous part of a character string, called a substring, can be accessed using the following syntax:

```
string ( [ starting-point ] : [ ending-point ] )
```

Where:

string is a string name or a CHARACTER literal constant except a Hollerith constant.

starting-point is the starting point of a substring of *string*.

ending-point is the ending point of a substring of *string*.

If absent, *starting-point* and *ending-point* are given the values one and the length of the *string*, respectively. A substring has a length of zero if *starting-point* is greater than *ending-point*.

For example, if `char_string` is the name of the string "`abcdefg`",

```
char_string(2:4)            ! is "bcd"
char_string(2: )            ! is "bcdefg"
char_string( :4)            ! is "abcd"
char_string( : )            ! is "abcdefg"
char_string(3:3)            ! is "c"
"abcdefg"(2:4)              ! is "bcd"
"abcdefg"(3:2)              ! is a zero-length string
```

1.5.8 Arrays

An array is a set of data, all of the same type and type parameters, arranged in a rectangular pattern of one or more dimensions. An array has a rank that is equal to the number of dimensions in the array; the maximum rank is 7. The array's shape is its extent in each dimension. The array's size is the number of elements in the array.

In the following example

```
integer, dimension (3,2) :: i
```

`i` has rank 2, shape (3,2), and size 6.

1.5.8.1 Array References

A whole array is referenced by the name of the array. Individual elements or sections of an array are referenced using array subscripts.

Syntax:

```
part [ % part ]...    or
part [ . part ]
```

Where:

part is the following syntax:

```
name [ ( subscript-list ) ]
```

name shall be an array if (*subscript-list*) is specified.

Using '%' operator or '.' operator is structure component reference (see "1.5.11.6 Structure Component").

subscript-list is a comma-separated list of

```
element-subscript       or
subscript-triplet       or
vector-subscript
```

element-subscript is a scalar INTEGER expression.

subscript-triplet is the following syntax:

```
[ element-subscript ] : [ element-subscript ] [ : stride ]
```

stride is a scalar INTEGER expression.

vector-subscript is a rank one INTEGER array expression.

The subscripts in *subscript-list* each refer to a dimension of the array. The left-most subscript refers to the first dimension of the array.

When *subscript-triplet* or *vector-subscript* is specified in *subscript-list*, *subscript-list* in other *part* shall not contain *subscript-triplet* and *vector-subscript*, and other name shall not be an array that is without (*subscript-list*).

If *name* without (*subscript-list*) is an array, other *name* shall not be an array that is without (*subscript-list*), and *subscript-triplet* and *vector-subscript* shall not be contained in *subscript-list* of other *part*.

1.5.8.2 Array Elements

If each subscript in an array subscript list is an *element-subscript*, then the array reference is to a single array element. Otherwise, more than one subscript in an array subscript list is a *subscript-triplet* or *vector-subscript*, it is to an array section (see "1.5.8.4 Array Sections").

1.5.8.3 Array Element Order

The elements of an array form a sequence known as array element order.

The position of an element of an array in the sequence is:

$$(1+(s_1-j_1))+((s_2-j_2)*d_1)+...+((s_n-j_n)*d_{n-1}*d_{n-2}*...*d_1)$$

Where:

s_i is the subscript in the ith dimension.

j_i is the lower bound of the ith dimension.

d_i is the size of the ith dimension.

n is the rank of the array.

For example, in the following code

```
integer, dimension(2,3) :: a
```

the order of the elements is `a(1,1)`, `a(2,1)`, `a(1,2)`, `a(2,2)`, `a(1,3)`, `a(2,3)`. When performing input/output on arrays, array element order is used.

1.5.8.4 Array Sections

You can refer to a selected portion of an array as an array. Such a portion is called an array section. An array section has a subscript list that contains at least one subscript that is either a *subscript-triplet* or a *vector-subscript*. Note that an array section with only one element is not a scalar.

1.5.8.5 Subscript Triplets

subscript-triplet is the following syntax:

```
[ element-subscript ] : [ element-subscript ] [ : stride ]
```

The first e*lement-subscript* is the lower bound of the array section, the second *element-subscript* is the upper bound, and *stride* means the increment between successive subscripts in the sequence. Any or all three can be omitted. If the lower bound is omitted, the declared lower bound of the dimension is assumed. If the upper bound is omitted, the upper bound of the dimension is assumed. If the *stride* is omitted, a stride of one is assumed.

Valid examples of array sections using subscript triplets are:

```
a(2:8:2)     ! a(2), a(4), a(6), a(8)
b(1,3:1:-1)  ! b(1,3), b(1,2), b(1,1)
c(:,:,:)     ! c
```

1.5.8.6 Vector Subscripts

Vector subscript is the following syntax:

```
vector-subscript
```

vector-subscript is a rank one INTEGER array expression.

A vector subscript designates a sequence of subscripts corresponding to the values of the elements of the expression.

Consider the following example:

```
integer :: vector(3) = (/3,8,12/)
real :: whole(3,15)
...
print*, whole(3,vector)
```

Here the array `vector` is used as a subscript of `whole` in the PRINT statement, which prints the values of elements (3,3), (3,8), and (3,12).

1.5.8.7 Array Reference with Substrings

A CHARACTER array section or array element can have a substring specifier following the subscript list. If an array section has a substring specifier, then the reference is an array section. A whole array cannot have a substring specifier.

For example,

```
character (len=10), dimension (10) :: string
string(3:8)(2:4) = 'abc'
```

assigns `'abc'` to the array section made up of characters 2 through 4 of rows 3 through 8 of the CHARACTER array `string`.

1.5.9 Dynamic Arrays

An array can be fixed in size at compile time or can assume a size or shape at run time in a number of ways:

- Allocatable arrays and array pointers can be allocated as needed with an ALLOCATE statement, and deallocated with a DEALLOCATE statement (see"1.5.9.1 Allocatable Arrays" and "1.5.9.2 Array Pointers"). Allocatable arrays and array pointers together are known as deferred-shape arrays.
- An assumed-shape array is a dummy array can assume a shape based on the shape of the corresponding actual argument (see "1.5.9.3 Assumed-Shape Arrays").
- An assumed-size array is a dummy array can assume a size based on the size of the corresponding actual argument (see "1.5.9.4 Assumed-Size Arrays").
- An automatic array is not dummy array and has bounds depend on the nonconstant expression (see "1.5.9.5 Automatic and Adjustable Arrays").

1.5.9.1 Allocatable Arrays

The ALLOCATABLE attribute can be given to an array, the array has ALLOCATABLE attribute is allocatable array. An allocatable array shall be a deferred-shape array (see "2.113 DIMENSION Statement"). An allocatable array shall be declared with deferred-shape specifier, ':', for each dimension. The deferred-shape specifier is the following syntax:

```
:
```

For example,

```
integer, allocatable :: a(:), b(:,:,:)
```

declares two allocatable arrays, a has rank one and b has rank three.

The bounds, and thus the shape, of an allocatable array are determined when the array is allocated with an ALLOCATE statement. Continuing the previous example,

```
allocate (a(3), b(1,3,-3:3))
```

allocates an array of rank one a, and size three and an array of rank three b, and size 21 with the lower bound -3 in the third dimension.

Memory for allocatable arrays is returned to the system using the DEALLOCATE statement.

1.5.9.1.1 Allocatable Array Association Status

The allocation status of an allocatable array is one of the following.

- Not currently allocated. An allocatable array with this status shall not be referenced or defined; it may be allocated with the ALLOCATE statement. Deallocating it causes an error condition in the DEALLOCATE statement. The ALLOCATED intrinsic returns .FALSE. for such an array.
- Currently allocated. An allocatable array with this status may be referenced, defined, or deallocated; Allocating it causes an error condition in the ALLOCATE statement. The ALLOCATED intrinsic returns .TRUE. for such an array.

The ALLOCATED intrinsic function (see "2.17 ALLOCATED Intrinsic Function") may be used to determine the allocation status.

If an object of derived type is created by an ALLOCATE statement, any allocatable ultimate components have an allocation status of not currently allocated unless the following.

- An object of derived type is specified in SOURCE= specifier, and
- The allocatable ultimate component is allocated.

An allocatable array with the SAVE attribute has an initial status of not currently allocated. If the array is allocated, its status changes to currently allocated. The status remains currently allocated until the array is deallocated.

An allocatable array that is a dummy argument of a procedure receives the allocation status of the actual argument with which it is associated on entry to the procedure. An allocatable array that is an ultimate component of a dummy argument of a procedure receives the allocation status of the corresponding component of the actual argument on entry to the procedure.

An allocatable array that does not have the SAVE attribute, that is a local variable of a procedure or un ultimate component thereof, that is not a dummy argument or a subobject thereof, and that is not accessed by use or host association, has a status of not currently allocated at the beginning of each invocation of the procedure. The status may change during execution of the procedure. If the array is not the result variable of the procedure or a subobject thereof and has a status of currently allocated when the procedure is exited by execution of a RETURN or END statement, it is deallocated.

An allocatable array that does not have the SAVE attribute and that is accessed by use association has an initial status of not currently allocated. The status may change during execution of the program. If the array has an allocation status of currently allocated when execution of a RETURN or END statement results in no scoping unit having access to the module, the status remains currently allocated.

1.5.9.2 Array Pointers

The POINTER attribute can be given to an array. An array pointer, like an allocatable array, shall be a deferred-shape array (see "2.113 DIMENSION Statement"). An allocatable array shall be declared with deferred-shape specifier, ':', for each dimension.
The deferred-shape specifier is the following syntax:

```
:
```

For example,

```
integer, pointer, dimension(:,:) :: c
```

declares a pointer array of rank two. An array pointer can be allocated in the same way an allocatable array can. Additionally, the shape of a pointer array can be set when the pointer becomes associated with a target in a pointer assignment statement. The shape then becomes that of the target.

```
integer, pointer, dimension(:,:) :: c
integer, target, dimension(2,4) :: d
c => d
```

In the above example, the array c becomes associated with array d and assumes the shape of d.

1.5.9.3 Assumed-Shape Arrays

An assumed-shape array is a nonpointer dummy array that takes its shape from the associated actual argument array. The lower bound of an assumed-shape array can be declared and can be different from that of the actual argument array. If the *lower-bound* is not present, the lower bound is 1.

An assumed-shape specification is

```
[ lower-bound ] :
```

for each dimension of the assumed-shape array.

For example,

```
integer :: a(3,4)
call zee(a)
contains
  subroutine zee(x)
    integer, dimension(-1:,:) :: x
    ...
  end subroutine zee
end
```

Here the dummy array x is an assumed-shape array, it assumes the shape of the actual argument a with a new lower bound for dimension one.

The interface for a procedure that has an assumed-shape dummy array shall be explicit (see "1.12.7.1 Explicit Interfaces").

1.5.9.4 Assumed-Size Arrays

An assumed-size array is a dummy array whose size is assumed from that of an associated actual argument. The rank and extents may differ for the actual and dummy arrays.

Declaration of an assumed-size array is the following syntax:

```
[ explicit-shape-spec-list ] [ lower-bound : ] *
```

An assumed-size array has no extent in its last dimension and no shape. The function name of an array-valued function shall not be declared as an assumed size array. An assumed-size array with INTENT(OUT) shall not be the following object:

- Derived type for which default initialization is specified, or
- Polymorphic, or
- Derived type which has an allocatable ultimate component, or
- Derived type which is finalizable.

You can not refer to an assumed-size array in a context where the shape of the array shall be known, such as in a whole array reference or for many of the transformational array intrinsic functions. A function result can not be an assumed-size array.

```
...
integer a
dimension a(4)
...
call zee(a)
...
end
subroutine zee(x)
integer, dimension(-1:*) :: x
...
```

In this example, the size of dummy array `x` is not known.

1.5.9.5 Automatic and Adjustable Arrays

A lower and upper bound of explicit-shape array may depend on the value of dummy argument, common block object, and the variable that is made accessible by use association or host association; it shall be a dummy argument, a function result, or an automatic array of a procedure. An automatic array is an explicit-shape array that is declared in a subprogram, is not a dummy argument, and has bounds that are nonconstant specification expressions.

Consider the following example:

```
subroutine foo(i, k)
  integer :: i,j,k
  dimension i(k,3), j(k)
```

Here the shapes of arrays `i` and `j` depend on the value of the dummy argument `k`. `j` is an automatic array.

1.5.10 Array Constructors

An array constructor is an unnamed array.

Syntax:

```
( / ac-spec / )    or
left-square-bracket ac-spec right-square-bracket
```

Where:

ac-spec is

```
[ type-spec:: ] [ ac-value-list ]
```

type-spec is a type specifier that is either an intrinsic type specifier or a derived type specifier.

See "2.469 Type Declaration Statement" for intrinsic type specifier, and see "1.5.11.8 Derived Type Specifier" for derived type specifier.

When the intrinsic type is specified for *type-spec*, the array component item expressions in one array constructor must be an intrinsic type that is compatible with the type variable of the type specifier and the type.

When the derived type is specified for *type-spec*, the array component item expressions in one array constructor are of that derived type and must have the same kind type parameter value as the type specified by the type specifier.

If *type-spec* is omitted, all array component item expressions in the array constructor must have the same length type parameter. In this case, the array constructor type and the type parameter are the same as the array component item expression.

A *type-spec* specification includes both the array constructor type and the type parameter. The array component item expressions in the array constructor must be compatible with the type and with the intrinsic assignment to the type parameter variable and type. These values are converted to the array constructor type parameter according to the intrinsic assignment rules.

left-square-bracket is a left-side square bracket. The left-side square bracket is '['.

right-square-bracket is a right-side square bracket. The right-side square bracket is ']'.

ac-variable is

```
expr                or
ac-implied-do
```

expr is an expression.

ac-implied-do is

```
( ac-value-list, ac-implied-do-control )
```

ac-implied-do-control is

```
ac-do-variable = scalar-int-expr, scalar-int-expr [ , scalar-int-expr ]
```

ac-do-variable is a named scalar INTEGER variable.

The variable specified in the array component implied DO *ac-do-variable* must not be the same as the array component implied DO *ac-do-variable* included in that array component implied DO.

scalar-int-expr is a scalar INTEGER expression.

An array constructor is a rank-one array. If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-value* is an array expression, the values of the elements of the expression, in array element order (see "1.5.8.3 Array Element Order"). If an *ac-value* is an *ac-implied-do*, it is expanded to form an *ac-value* sequence under the control of the *ac-do-variable*, as in the DO construct (see "2.114 DO Construct").

If *type-spec* is omitted, the array component item expression type and the kind type parameter must be the same in one array constructor.

When the intrinsic type is specified in the type specifier, the array component item expression type in one array constructor must be an intrinsic type that is compatible with the type specifier.

When the derived type is specified in the type specifier, the array component item expression type in one array constructor is the derived type and must have the same kind type parameter value as the type specified by the type specifier.

If the type specifier is omitted, all array component item expressions in the array constructor must have the same length type parameter.

```
integer, dimension(3) :: a, b=(/1,2,3/), c=(/(i, i=4,6)/)
a = b + c + (/7,8,9/)
!! a is assigned (/12,15,18/)
```

An array constructor can be reshaped with the RESHAPE intrinsic function and can then be used to initialize or represent arrays of rank greater than one.

For example,

```
integer,dimension(2,2) :: a = reshape((/1,2,3,4/),(/2,2/))
```

assigns `(/1,2,3,4/)` to a in array-element order after reshaping it to conform with the shape of `a`.

1.5.11 Derived Types

Derived types are user-defined data types based on the intrinsic types, INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER. A scalar data object of derived type is called a structure.

1.5.11.1 Derived Type Definition

When a data entity is declared explicitly to be of a derived type, the derived type shall have been defined previously in the scoping unit or be accessible there by use or host association. The component of each data entity that is of type derived type is declared to be of the types specified by the corresponding component definition statements of the derived type definition.

Syntax:

```
TYPE [ [ , type-attr-spec-list ] :: ] type-name [ ( type-param-name-list ) ]
    [ type-param-def-statement ] ...
    [ private-or-sequence ] ...
    [ component-def-statement ] ...
```

```
    [ type-bound-procedure ] ...
END TYPE [ type-name ]
```

Where:

type-attr-spec-list can specify PUBLIC, PRIVATE, EXTENDS (parent type name), ABSTRACT, and BIND(C). PUBLIC and PRIVATE cannot be specified concurrently. The parent type must be the name of an extensible type (see "1.5.11.4 Type Extension") defined previously. If the type definition includes a deferred binding, or if a deferred binding is inherited, ABSTRACT must be specified. If ABSTRACT is specified, that type must be extensible. If EXTENDS is specified, SEQUENCE cannot be specified.

type-name is the name of the derived type being defined. Specify the type attribute specification (see "1.5.11.2 Derived Type Parameter") in *type-param-name-list.*

type-param-def-statement is the type parameter definition statement (see "1.5.11.2 Derived Type Parameter").

private-or-sequence-statement is

```
PRIVATE        or
SEQUENCE
```

component-def-statement is

```
component-data-def-statement        or
component-proc-def-statement
```

component-def-statement is the data component declaration statement. (see "2.469 Type Declaration Statement"), only POINTER, ALLOCATABLE, CONTIGUOUS, DIMENSION, PUBLIC and PRIVATE is permitted for *attr-spec*. If the POINTER or ALLOCATABLE attribute is specified, each array specifier shall be a *deferred-shape-spec-list*. If the POINTER and ALLOCATABLE attribute is not specified, each array specifier shall be an *explicit-shape-spec-list*. POINTER and ALLOCATABLE shall not both appear in the same *component-def-statement*. If the CONTIGUOUS attribute is specified, the component shall be an array with the POINTER attribute.

component-proc-def-statement is a component procedure definition statement, is the following:

```
PROCEDURE ( [ proc-interface ] ), proc-component-attr-spec-list :: proc-decl-list
```

proc-interface is a procedure interface.

proc-component-attr-spec-list is a list of procedure component attribute specification.

proc-decl-list is a list of procedure declaration.

POINTER, PASS [(argument name)], NOPASS, PUBLIC, and PRIVATE can be specified in the procedure component attribute specifier, and POINTER must always be specified. If the procedure pointer component has an implicit interface or has no arguments, NOPASS must be specified. If PASS [(argument name)] is specified, the interface must have a dummy argument that sets the argument name. The dummy argument name is the passed-object dummy argument of the procedure pointer component. PASS and NOPASS cannot be specified concurrently in the same procedure component attribute specifier list.

type-bound-procedure is a type-bound procedure (see "1.5.11.3 Type-Bound Procedure").

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

If the SEQUENCE statement is present, the type is a sequence type. The storage of the components for sequence type in memory as definition order. If SEQUENCE is present, all derived types specified in component definitions shall be sequence types.

If all of the ultimate components are of type default INTEGER, default REAL, double precision REAL, default COMPLEX, or default LOGICAL and are not pointers or allocatable variables, the type is a numeric sequence type.

If all of the ultimate components are of type default CHARACTER and are not pointers or allocatable variables, the type is a character sequence type.

The PRIVATE attribute and statement in the type definition can be specified only in the type definition in the module declaration part. Even if the type has the PUBLIC attribute, the component name of that PRIVATE attribute can be referenced only within the module with that type definition. If that type has the PRIVATE attribute, the structure constructor can be referenced only within the module with that type definition.

A derived type can be declared by STRUCTURE statement.

Syntax:

```
STRUCTURE [ / type-name / ] [ component-list ]
    struct-elm-def-statement
    [ struct-elm-def-statement ] ...
END STRUCTURE
```

If derived type definitions using STRUCTURE statements are nested, only the *type-name* argument in the inner definition may be omitted. The *component-list* can be specified only in the inner STRUCTURE statement of the nested definition. The *component* arguments are structure names if a derived type being defined.

struct-elm-def-statement is the following syntax:

```
component-def-statement     or
union-declaration           or
structure-definition
```

union-declaration is the following syntax:

```
UNION
    map-declaration
    [ map-declaration ] ...
END UNION
```

map-declaration declares the block that shares same storage.

map-declaration is the following syntax:

```
MAP
    struct-elm-def-statement
    [ struct-elm-def-statement ] ...
END MAP
```

Each *map-declaration* in the *union-declaration* shares the same storage. The size of the storage space occupied by a union is equal to the maximum size of the map declarations.

structure-definition is a derived type definition by STRUCTURE statement.

The derived type defined by STRUCTURE statement is a sequence type.

If *initialization-expr* appears for a nonpointer component, that component in any object of the type has initial value, also component of allocatable variable and automatic object have initial value when they are allocated. If *initialization-expr* appears for a pointer component, that component in any object of the type has default initial status; disassociated. They are called default initialization.

initialization-expr shall not appear for an allocatable variable component.

If a component is of a type for which default initialization is specified for component, the default initialization specified *initialization-expr* overrides the default initialization specified for that component. When one initialization overrides another it is as if only the overriding initialization were specified. Explicit initialization in a type declaration statement overrides default initialization.

In the following example,

```
type coordinates
  real :: latitude=0.0, longitude=0.0
end type coordinates

type place
  character(len=20) :: name
  type(coordinates) :: location
end type place

type link
  integer :: j
  type (link), pointer :: next=>null()
end type link
```

type `coordinates` is a derived type with two REAL components: `latitude` and `longitude`: and each component has initial value 0.0. Type `place` has two components: a CHARACTER of length twenty, `name`, and a structure of type `coordinates` named

`location`. Type `link` has two components: an INTEGER, `j`, and a structure of type `link`, named `next`, that is a pointer to the same derived type. A component structure can be of the same type as the derived type itself only if it has the POINTER attribute and has initial status: disassociated.

Two data entities have the same type if they are declared with reference to the same derived type definition. The definition may be accessed from a module or from a host scoping unit. Data entities in different scoping unit also have the same type if they are declared with reference to different derived type definitions that have the same name, have the SEQUENCE property, have the BIND(C) property, and have components that do not have PRIVATE accessibility and agree in order, name, and attributes.

1.5.11.2 Derived Type Parameter

The type parameter definition statement declares the name of the type parameter in the derived type definition.

The type parameter definition statement has the following format:

Syntax:

```
INTEGER [ kind-selector ] , type-param-attr-spec :: type-param-decl-list
```

Where:

kind-selector is a kind type parameter.

type-param-attr-spec is an attribute specifier of the type parameter.

type-param-decl-list is a declaration list of type parameters,

that is the following syntax:

```
type-param-name [ = scalar-init-initialization-expr ]
```

type-param-name is a name of the type parameter.

scalar-init-initialization-expr is a scalar INTEGER initialization expression.

The type parameter name in the type parameter definition statement of the derived type definition must be included in the TYPE statement of the derived type definition.

The type parameter names in the TYPE statement of a derived type definition must be included in the type parameter definition statement of the derived type definition.

The type parameter attribute specifier has the following format:

```
KIND    or
LEN
```

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

The KIND attribute specifier specifies kind type parameter and the LEN attribute specifier specifies length type parameter. A length type parameter may be specified in specification expressions within the derived type definition for the type, but it must not be specified in initialization expressions. If component initialization appears, every type parameter and array bound of the component must be constant expression.

The type parameter order of an extended type consists of the type parameter order of its parent type followed by any additional type parameters in the order of the type parameter list in the derived type definition.

The dummy argument of derived type having length type parameter cannot have INTENT(OUT) attribute.

Parameterized derived type definition, object declaration, or component declaration cannot be specified in scope of BLOCK construct.

Structure component to the right of part references with nonzero rank cannot specify length type parameter in array bound or character length specification.

The object or the expression of a derived type that has ultimate component where length type parameter is used cannot be specified in an input/output list item or NAMELIST statement.

Intrinsic procedures with parameterized derived type as result type cannot be specified if it is passed as actual argument and corresponding dummy argument has assumed type length parameters.

The type of the allocate object cannot be parameterized derived type when the MOLD specifier is specified for the ALLOCATE statement.

The expression of parameterized derived type cannot be specified as selector.

Example for derived type parameter:

```
type struct(dim,lg)
  integer, kind :: dim
  integer, len :: lg
  real :: array(dim)
  character(len=lg) :: ch
end type
```

1.5.11.3 Type-Bound Procedure

The type bound procedure specifies a procedure for operating the derived type in the TYPE statement in the derived type definition.

The type bound procedure has the following format:

Syntax:

```
CONTAINS statement
[ binding-private-stmt ]
proc-binding-stmt
[ proc-binding-stmt ] ...
```

Where:

binding-private-stmt is a binding private statement, is the following:

```
PRIVATE
```

The binding PRIVATE statement can be specified only when the type definition is in the module specification.

proc-binding-stmt is a procedure binding statement.

The procedure binding statement must be either a specific binding, a generic binding, or a final binding.

A specific binding has the following format:

```
PROCEDURE[(interface-name)][[,binding-attr-list]::]binding-assignment-list
```

interface-name is a name of the interface.

binding-attr-list is an attribute list of binding.

binding-assignment-list is a comma-delimited binding procedure name declaration list.

The binding procedure name declaration has the following format:

```
binding-name [ => procedure-name ]
```

binding-name is a name of binding.

procedure-name is a name of procedure.

When "=> *procedure-name*" is specified, the two consecutive delimiting colons cannot be omitted.

When "=> *procedure-name*" is specified, the interface name cannot be specified.

The procedure name must be a module procedure name that can be accessible, or an external procedure name that has an explicit interface.

If "=>*procedure-name*" and an interface name are both not specified, it is the same as when a binding name is specified in the "=>*procedure-name*" procedure name.

A generic binding has the following format:

```
GENERIC [ , access-attr ] :: generic-spec=> binding-name-list
```

access-attr is an access attribute.

generic-spec is a generic specifier.

binding-name-list is a list of binding name.

These generic bindings must explicitly or implicitly specify in the module specification part the same accessibility attribute as all other generic bindings of the same generic specifier in that derived type.

The binding name of the binding name list must be the name of the specific binding of that type.

If the generic specifier is not a generic name, these specific bindings must have a passed-object dummy argument (see "1.12.7.3 Generic Interfaces").

If the generic specifier is OPERATOR(user-defined-operator), see "1.12.7.3.2 Defined Operations".

If the generic specifier is OPERATOR(=), see "1.12.7.3.3 Defined Assignment".

An *access-attr* has the following format:

```
PUBLIC    or
PRIVATE
```

An *access-attr* can specify into module definition part.

A binding attribute has the following format:

```
PASS [ ( arg-name ) ]    or
NOPASS                   or
NON_OVERRIDABLE          or
DEFERRED                 or
access-attr
```

arg-name is an argument name.

access-attr is an access attribute.

The same binding attribute can be specified only once in one binding attribute list.

If that binding interface does not have a dummy argument of defined type, NOPASS must be specified.

When PASS(*arg-name*) is specified, a dummy argument with an argument name must be specified in that binding interface. The dummy argument name is the passed-object dummy argument of type bound procedure.

PASS and NOPASS cannot be specified concurrently in one binding attribute list.

NON_OVERRIDABLE and DEFERRED cannot be specified concurrently in one binding attribute list.

If an interface name is specified, DEFERRED must be specified. In this case, the interface name must be specified.

The DEFERRED attribute must be specified only when the type bound procedure to be overridden is deferred.

An inherited binding with the NON_OVERRIDABLE attribute cannot be overridden.

Each binding of a procedure binding statement specifies a type bound procedure. A type bound procedure can have a passed-object dummy argument. A generic binding specifies the generic interface of the type binding for each specific binding. A binding with the DEFERRED attribute is a deferred binding, which can be specified only within an abstract type definition.

A type bound procedure can be identified by a binding name within the scope of the type definition. This name is used as the binding name for a specific binding, and the generic name in the generic specifier for a generic binding is the generic name. A generic binding that is different from a final binding or a generic name in a generic specifier does not have a binding name, nor does a final binding.

A specific binding interface is the interface of the procedure specified by the procedure name or the interface name.

Example of a type and type bound procedure.

```
module mod
type two_real
  real :: x,y
  contains
    procedure, pass :: two_real_fun=> fun
end type two_real
contains
  real function fun (a, b)
    class (two_real), intent (in) :: a, b
    fun = a%x + b%x
```

```
  end function fun
end module
```

The same generic specifier can be specified in multiple generic bindings in one derived type definition. Added generic bindings with the same generic specifier extend the generic interface.

A generic type bound procedure name cannot be the same name as a specific type bound procedure name with the same type.

When a type definition includes a binding PRIVATE statement, the implicit accessibility of the type procedure binding is private. If a binding PRIVATE statement is not included, it is public. The procedure binding accessibility can be specified explicitly.

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

A final binding has the following format:

```
FINAL [ :: ] final-subroutine-name-list
```

final-subroutine-name-list is a final subroutine name.

A *final-subroutine-name* is the name of a module procedure that has only one dummy argument. That argument cannot be omitted, and is a variable of the derived type being defined, but not a pointer, allocatable variable, or polymorphic. The dummy argument cannot have INTENT(OUT) or VALUE attribute.

A *final-subroutine-name* must not be one previously specified for that type. A final subroutine must not have a dummy argument with the same kind type parameters and rank as the dummy argument.

A derived type is finalizable when the derived type has the following conditions,

- It has any final subroutines, or
- It has any nonpointer, nonallocatable component whose type is finalizable.

Finalization occurs when following conditions,

- When a pointer is deallocated its target is finalized, or
- An allocatable object is deallocated, or
- An object is finalized in RETURN statement or END statement when the following conditions.
  - The program unit is not main program, and
  - The object does not have SAVE attribute, and
  - The object is not a dummy argument, and
  - The object is not a function result, and
  - The object does not have POINTER attribute, and
  - The object is not defined in a module
- If an executable construct references a function, the result is finalized after execution of the innermost executable construct.
- If an executable construct references a structure constructor, the entity created by the structure constructor is finalized after execution of the innermost executable construct.
- When a procedure is invoked, a nonpointer, nonallocatable object that is an actual argument associated with an INTENT(OUT) dummy argument is finalized.
- When an intrinsic assignment statement is executed, variable is finalized after evaluation of expression and before the definition of variable.

1.5.11.4 Type Extension

The extendable type is a nonsequence derivative type that does not have the BIND attribute.

The extended type has the EXTENDS attribute. Its parent type is specified in the EXTENDS attribute.

The abstract type has the ABSTRACT attribute.

Example of type extension:

```
type a_type
  real :: x, y
end type a_type
type, extends(a_type) ::a_extend_type ! extended type for a_type
real:: z
end type a_extend_type
```

1.5.11.4.1 Inheritance

The extended type extends the parent type. The extended type includes all type parameters of the parent type, all components, and procedure bindings that do not override or finalize. The extended type holds all attributes held by the parent type. The extended type derived type definition can add type parameters, components, and procedure bindings.

An extended type has a scalar, nonpointer, nonallocatable, parent component with the type and type parameters of the parent type. The name of this component is the parent type name. The component has the accessibility attribute of the parent type. The component of the parent component is inheritance associated with the corresponding component inherited from the parent type. The component declared in the extended type and the type parameter name cannot be the same as the accessible component and the type parameter name of the parent type.

Example of inheritance:

```
type t1                ! base type
  real :: x, y
end type t1
type, extends(t1) ::t2 ! extended type
                       ! component x,y,t1 are inherited.
  real :: z
end type t2
type(t2) :: a
```

1.5.11.4.2 Overriding a Type Bound Procedure

When a binding other than a generic binding specified in the type definitions has the same binding name as the parent type binding, the binding specified in the type definition overrides the binding from the parent type.

When the generic binding specified by the type definition has the same generic specifier as an inherited binding, the generic interface is extended. The overriding binding procedure and the overridden binding procedure must meet all of the following conditions:

- Both must have a passed-object dummy argument, or both must not have one.
- If the overridden bound procedure is a pure procedure, the overriding bound procedure must also be a pure procedure.
- Both are elemental procedures, or neither can be an elemental procedure.
- Both must have the same number of dummy arguments.
- If a passed-object dummy argument appears, the name and position must correspond.
- Dummy arguments in corresponding positions must have the same name and characteristics unless they are the passed-object dummy argument type.
- Both must be subroutines, or be functions with the same result and characteristics.
- If the overridden bound procedure has the PUBLIC attribute, the overriding bound procedure cannot have the PRIVATE attribute.

Example of type bound procedure overriding:

```
module mod
type t1
  real :: x, y
  contains
  procedure, pass :: fun => afun
end type t1
type, extends(t1) ::t2
  real:: z
```

```
  contains
  procedure, pass :: fun => bfun
end type t2
contains
  function afun (a, b) result(r)
    class (t1), intent (in) :: a, b
    real :: r
    r = a%x + b%x + a%y + b%y
  end function afun
  function bfun (a, b) result(r)
    class (t2), intent (in) :: a
    class (t1), intent (in) :: b
    real :: r
    select type(b)
    type is(t2)
      r = a%x + b%x + a%y + b%y + a%z +b%z
    end select
  end function bfun
end module mod
```

1.5.11.5 Declaring Variables of Derived Type

Variables of derived type are declared with the type declaration statement with TYPE (*type-name*) *type-specifier*. The following are examples of declarations of variables for each of the derived types defined above:

```
type(coordinates) :: my_coordinates
type(place) :: my_town
type(place), dimension(10) :: cities
type(link) :: head
```

1.5.11.6 Structure Component

Components of a structure are referenced using the percent sign '%' operator or '.' operator.

Syntax:

```
part % part [ % part ] ...        or
part . part [ . part ] ...
```

Where:

part is the following syntax:

```
name [ ( subscript-list ) ] [ ( substrings ) ]
```

If (*subscript-list*) is specified, *name* must be an array. See "1.5.8.1 Array References" for reference of arrays. If (*substrings*) is specified, *name* must be of type character. See "1.5.7 Substrings" for reference of substring.

Each *name* except the rightmost shall be of derived type, and except the leftmost shall be the name of a component of the derived type definition of the type of the preceding *name*.

To perform a procedure reference of structure components, the name at the right must be the procedure pointer component name.

When uses '.' operator for structure component reference, component name shall not be the same as 'TRUE', 'FALSE', 'EQ', 'NE', 'GT', 'GE', 'LT', 'LE', 'NOT', 'AND', 'OR', 'EQV' nor 'NEQV', also shall not be the same as user-defined operator.

The following are examples of structure component reference for each of the variable declared above:

```
my_coordinates%latitude
my_town%location%latitude
cities(:,:)%name
cities%name
cities(1,1:2)%location%latitude
```

`my_coordinates%latitude` refers `latitude` in the structure `my_coordinates`.

`my_town%location%latitude` refers `latitude` in type `coordinates` in structure `my_town`.

`cities(:,:)%name` refers `name` for all element of `cities`. It is the same meaning as `cities%name`.

`cities(1,1:2)%coordinates%latitude` refers element `latitude` of type `coordinates` for elements `(1,1)` and `(1,2)` only of `cities`.

1.5.11.7 Enumeration Bodies and Enumerators

An enumeration body definition specifies a pair consisting of an enumeration body and the enumerator of the corresponding default INTEGER type.

An enumeration has the following format:

```
ENUM , BIND ( C )
  ENUMERATOR [ :: ] named-constant [ = scalar-int-initialization-expr ]
  [ ENUMERATOR [ :: ] named-constant [ = scalar-int-initialization-expr ] ]...
END ENUM
```

named-constant is a list of enumerators that is a named constant of default INTEGER type.

scalar-int-initialization-expr is a scalar INTEGER initialization expression.

When = is specified in an enumerator, two consecutive delimiting colons must be specified before the enumerator list.

The PARAMETER attribute is implicitly specified in enumerators. Enumerators are defined according to the intrinsic assignment rules, and values are determined as follows:

1. If a scalar INTEGER initialization expression is specified, the enumerator value is the result value of the scalar integer initialization expression.
2. If a scalar INTEGER initialization expression is not specified and the enumerator is the first enumerator in the enumeration body definition, the enumerator value is 0.
3. If a scalar INTEGER initialization expression is not specified and the enumerator is not the first enumerator in the enumeration body definition, 1 is added to the preceding enumerator in the definition, and the result becomes the enumerator value.

Example of enumerator is the following:

```
enum, bind(c)
  enumerator :: desktoppc = 2007, notepc = 2009
  enumerator netbookpc
end enum
```

The following example is same as above.

```
integer, parameter :: desktoppc = 2007, notepc = 2009, netbookpc = 2010
```

1.5.11.8 Derived Type Specifier

A derived type specifier specifies derived type and type parameters.

Syntax:

```
type-name [ ( type-param-spec-list ) ]
```

Where:

type-name is the name of the derived type. *type-name* must have been defined previously in the scoping unit or be accessible there by use or host association.

type-param-spec is:

```
[ keyword = ] type-param-value
```

(*type-param-spec-list*) can specify only if the type is parameterized.

There must 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 must be a *type-param-spec* corresponding to that type parameter.

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*.

Each *keyword* is the name of a parameter of the type.

type-param-value is:

```
scalar-int-expr     or
*                   or
:
```

The type param value for a kind type parameter must be an initialization expression.

A colon may be used as a type parameter value only in the declaration of an entity or component that has the POINTER or ALLOCATABLE attribute.

A colon as a type parameter value specifies a deferred type parameter. The deferred type parameter is a length type parameter whose value can change during execution of the program.

An asterisk as a type parameter value specifies an assumed type parameter. The asterisk can 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.

1.5.12 Structure Constructors

A structure constructor is a scalar value of derived type to be constructed from a sequence of values, one value for each component of the derived type.

Syntax:

```
derived-type-spec ( [ expr-list ])
```

Where:

derived-type-spec is a derived type specifier (see "1.5.11.8 Derived Type Specifier"). The abstract type cannot be specified as the derived type name.

The type parameter of derived type specifier can be specified only if the derived type has the type parameters. When derived type has the type parameter it cannot be omitted.

expr-list is a list of component specifier that has the following format:

```
[ keyword= ] component-data-source
```

keyword is a component keyword that is a name of component.

component-data-source is a component data source that has the following format:

```
expr            or
data-target     or
proc-target
```

expr is for an expression that specifies a value of a component of a derived type. *data-target* is for a data target. *proc-target* is for a procedure target.

Two or more component specifications, including components associated by inheritance, cannot be specified in one derived type component. If the component has no implied initial value, it must have a component specification. "keyword=" can be omitted from the component specification only if "keyword=" is omitted from each of the preceding component specifications in the structure constructor.

All components of a type with a specified type name and component specification must be able to be referenced by the scoping unit that contains the structure constructor.

If the type name is the same as a generic name, the component specification list cannot be a valid actual argument specifier list of a function reference resolved as a generic reference.

A data target must correspond to a pointer component that is not a procedure pointer. A procedure target must correspond to a component that is a procedure pointer.

A data target must have the same rank as the corresponding component.

If keyword= is not present, the corresponding component data source is assigned according to the component order. If keyword= is present, it is assigned to the component named by the keyword. For components that are not pointers, the type and type parameter at the time of declaration of that component and expression must be compatible with the variables and expressions in the intrinsic assignment statement. For components that are not pointers or allocation components, the expression shape must be compatible with the component shape.

For a pointer component, the corresponding component data source must be a procedure target or a data target permitted by the pointer assignment statement.

If a derived type component is an allocation element, the corresponding component data source must be evaluated as an intrinsic function NULL() with the argument omitted (see "2.350 NULL Intrinsic Function"), an allocated variable with the same rank, or an object with the same rank. If the expression is a reference to the intrinsic function NULL(), the corresponding component of the constructor has a status of not currently allocated. If the expression is an allocatable entity, the corresponding component of the constructor has the same allocation status as that allocatable entity and, if it is allocated, the same bounds and value. With any other expression that evaluates to an array the corresponding component of the constructor has an allocation status of currently allocated and has the same bounds and value as the expression.

When a component with an implicit initial value set does not have a corresponding component data source, an implicit initial value is set for that component.

A structure constructor cannot be specified before the referenced type is defined.

Example:

```
type my_type              ! derived-type definition
  integer :: i,j
  character(len=40) :: string
end type my_type
type (my_type) :: a       ! derived-type declaration
a = my_type (4, 5.0*2.3, 'abcdefg')
```

In this example, the second expression in the structure constructor is converted to type default INTEGER when the assignment is made.

1.5.12.1 Component Keyword

When a component keyword '*keyword*=' is specified, the expression is associated with the component specified at keyword regardless of the position in the structure constructor list.

If a component keyword is not specified, the expression is associated with the component at the position corresponding to the derived type component list.

If the expression is not associated with the component at the position corresponding to the component list, and if there is a component keyword specification for the preceding expression in the expression list, a component keyword must be specified.

Example of component keyword:

```
type ty1
  integer :: a,b,c
end type
type(ty1):: t
t = ty1(c=1,b=2,a=3)
end
```

In this example, the `ty1(c=1,b=2,a=3)` is same as `ty1(3,2,1)`

1.5.12.2 Components for which an Expression can be Omitted

If a component has an implicit default initialization, the corresponding expression can be omitted.

If an expression is validly omitted but the component in question is not the last component in the component list, a component keyword specification is required for subsequent components in the actual argument list.

Example of an optional component:

```
type ty1
  integer :: a = 1
  integer :: b
  integer :: c = 1
```

```
end type
type (ty1) :: t
t = ty1(b = 3)    ! a and c are omitted,
                  ! and must specify component keyword for b.
t = ty1(2, 3)     ! c is omitted,
                  ! and no require component keyword for a or b.
t = ty1(b = 3, c = 8)
                  ! a is omitted,
                  ! but the component keywords
                  ! must be specified for b and c.
end
```

1.5.12.3 Extended Type of Derived Type

If a derived type is an extended type from a parent type, the parent type variable with a parent type name is treated as the top component. If there is no expression associated with the parent type variable in the structure constructor, the expression corresponding to the parent type component must be specified.

Example:

```
type ty1
  integer :: a
end type
type, extends(ty1) :: ty2
  integer :: b
end type
type (ty1) :: t1
type (ty2) :: t2
  ...
t2 = ty2(1, b = 1)
t2 = ty2(ty1 = ty1(1), b = 1)
t2 = ty2(a = 1, b = 1)
end
```

1.5.13 Class

A polymorphic data element can have a different type during program execution. The polymorphic data element type during execution is the type during program execution. The data element declared type is the explicitly or implicitly declared type.

The CLASS specifier declares a polymorphic object. If the CLASS specifier contains a type name, that type becomes the polymorphic object declared type.

The CLASS(*) specifier declares an unlimited polymorphic object. Unlimited polymorphic data elements are not declared as objects having a type. In addition, the declared type is not the same for all other elements, including other unlimited polymorphic data elements.

If a data element is not polymorphic, the type is compatible only with data elements that have the same declared type. If a polymorphic data element is not unlimited polymorphic, the data elements with compatible types may have the same declared type or may have its extended type. Unlimited polymorphic data elements do not have a declared type. However, all data elements and types are compatible. If "the type is compatible" between a data element and a particular type, this means that data elements of that type are compatible with the type. If two data element types are "not compatible", both types are mutually incompatible.

To make a data element and another element TKR compatible means that the data element is compatible with the target data element and type, that both have the same kind type parameter, and that they also have the same rank.

A polymorphic allocate object can allocate an object of any compatible type. A polymorphic pointer or polymorphic dummy argument can associate an object of a compatible type during program execution.

The type at execution of an allocated polymorphic allocate object is the allocation time type.

At execution, the type of a polymorphic pointer in the associated state is the target dynamic type.

At execution, the type of a polymorphic dummy argument that is not an allocate object or a pointer is the associated actual argument dynamic type.

At execution, the type of an allocate object not in the allocated state or of an empty pointer is the declared type.

At execution, the type of an object that has an association name is the dynamic type of the selector that associated it.

The type at execution of an object that is not polymorphic is the declared type.

1.5.14 Pointer

A pointer is a variable that has the POINTER attribute. There are "data pointers" and "procedure pointers".

A data pointer is not referenced or defined unless it is associated with a target object that can be referenced or defined.

A procedure pointer is a procedure that has the EXTERNAL attribute and the POINTER attribute. It is not referenced unless the target procedure and pointer are associated.

1.5.14.1 Pointer Association

A data pointer is associated with a target by pointer assignment (see "2.363 Pointer Assignment Statement") or allocation (see "2.16 ALLOCATE Statement"). A target shall be a variable that has the TARGET attribute or shall be a pointer. A pointer that is associated with a target can refer or define, any reference to the pointer applies to the target. If the pointer is polymorphic, the dynamic type is the target dynamic type.

Pointer assignment (see "2.363 Pointer Assignment Statement") can be used to associate a procedure pointer with a dummy procedure that is not a module procedure, intrinsic procedure, or a procedure pointer.

1.5.14.2 Pointer Association Status

A pointer association status is one of associated, disassociated, or undefined. The ASSOCIATED intrinsic function (see "2.29 ASSOCIATED Intrinsic Function") may be used to determine the association status.

Unless a pointer is initialized (explicitly or by default), it has an initial association status of undefined. An initialized pointer has an association status of disassociated. If the association status of a pointer is disassociated or undefined, the pointer shall not be referenced or deallocated. Whether its association status, a pointer always may be nullified by NULLIFY statement, allocated by ALLOCATE statement, or pointer assigned. A nullified pointer is disassociated. When a pointer is allocated, it becomes associated. When a pointer is pointer assigned, its association status is determined by its target.

1.5.14.3 Declaring Data Pointers and Targets

A data pointer can be declared in a type declaration statement with the POINTER *attr-spec* or in a POINTER statement. A target can be declared in type declaration statement with the TARGET *attr-spec* or in a TARGET statement. When declaring an array to be a pointer, you shall declare the array with a deferred-shape array.

Example:

```
integer, pointer :: a, b(:,:)
integer, target :: c
a => c                ! pointer assignment statement
                      ! a is associated with c
allocate (b(3,2))     ! allocate statement
                      ! an unnamed target for b is
                      ! created with the shape (3,2)
```

In this example, an explicit association is created between a and c through the pointer assignment statement. Note that a has been previously declared a pointer, c has been previously declared a target, and a and c agree in type, kind, and rank. In the ALLOCATE statement, a target array is allocated and b is made to point to it. The array b was declared with a deferred shape, so that the target array could be allocated with any rank two shape.

1.5.15 Implied Shape Array

An implied-shape array is a named constant that takes its shape from the initialization expression in its declaration. If the lower bound is not present, it becomes 1.

An implied-shape specification is

```
[ lower-bound  : ] *
```

for each dimension of the implied-shape array.

An implied-shape array must be a named constant.

For example,

```
integer,dimension(0:*),parameter :: x=[1,2,3]
```

Here the named constant x is an implied-shape array, it takes the shape from initialization expression and lower bound is 0 and upper bound is 2 for dimension one.

1.6 Expressions

An expression is formed from operands, operators, and parentheses. Evaluation of an expression produces a value with a type, type parameters (kind and, if CHARACTER, length), and a shape.

Some examples of valid Fortran expressions are

```
5
n
(n+1)*y
"Fujitsu " // 'Fortran ' // text(1:23)
(-b + (b**2-4*a*c)**.5) / (2*a)
b%a - a(1:1000:10)
sin(a) .le. .5
l .operator. r + .operator. m
```

The last example uses defined operations `.operator.` (see "1.12.7.3.2 Defined Operations").

All array-valued operands in an expression shall have the same shape. A scalar is conformable with an array of any shape. Array-valued expressions are evaluated element-by-element for corresponding elements in each array and a scalar in the same expression is treated like an array where all elements have the value of the scalar.

For example, the expression

```
a(2:4) + b(1:3) + 5
```

would perform

```
a(2) + b(1) + 5
a(3) + b(2) + 5
a(4) + b(3) + 5
```

Expressions are evaluated according to the rules of operator precedence (see "1.6.3 Intrinsic Operations"). If there are multiple contiguous operations of the same precedence, exponentiation is evaluated from right to left, and other operations are evaluated from left to right. Parentheses can be used to enforce a particular order of evaluation.

For example,

```
a + (b-c)
```

operation `b-c` is evaluated first, then addition `a` with the result of is `(b-c)` evaluated.

Unary addition or subtraction cannot be specified following ** , / , * , + , or - operator. The unary addition or subtraction is necessary enclosed by parentheses.

For example, the incorrect expressions are

```
K + -1    ! Fix to K+(-1)
K ** -2   ! Fix to K**(-2)
```

1.6.1 Specification Expression

A specification expression is a scalar INTEGER expression for use in specifications such as length type parameters and array bounds. The scalar INTEGER expression is a restricted expression.

The restricted expression is an expression in which each operation is intrinsic and the expression must comprise one of the items below. However, all subscripts, array section subscripts, substring start positions, substring end positions, and type parameter values in these must be restricted expressions.

- A constant or subobject of a constant.
- A dummy argument that has neither the OPTIONAL nor the INTENT(OUT) attribute.
- An object that is in a common block.
- An object that is made accessible by use association or host association.
- An array constructor where each element and each scalar INTEGER expression of DO type repeat is a restricted expression.
- A structure constructor where each component is a restricted expression.
- A specification inquiry function argument is the follow either
    1. A restricted expression, or
    2. A variable whose properties inquired about are not the following
        - Dependent on the upper bound of the last dimension of an assumed size array.
        - Deferred.
- A reference to any other standard intrinsic function where each argument is a restricted expression.
- A reference to a specification function where each argument is a restricted expression.
- A type parameter of the derived type being defined.
- A restricted expression enclosed in parentheses.

A specification inquiry shall be the following reference

- An array inquiry function
- The bit inquiry function BIT_SIZE
- The character inquiry function LEN or NEW_LINE
- The kind inquiry function KIND
- A numeric inquiry function
- A type parameter inquiry
- An IEEE inquiry function

A function is a specification function if it is a pure function, is not a standard intrinsic function, is not an internal function, is not a statement function, and does not have a dummy procedure argument.

Evaluation of a specification expression shall not directly or indirectly cause a procedure defined by the subprogram in which it appears to be invoked.

1.6.2 Initialization Expression

An initialization expression is a constant expression that can be evaluated at compile time. An initialization expression must comprise one of the items below. However, all subscripts, array section subscripts, substring start positions, substring end positions, and type parameter values in these must be initialization expressions.

- Array constructor

  All elements, DO type repeat shape specifications, and strides in the array constructor must be initialization expressions.

- Structure constructor

  All component specifications corresponding to allocate components in the structure constructor must be transformational intrinsic function NULL references, and all other component specifications must be initialization expressions.

- Elemental intrinsic function reference

  All references must be initialization expressions.

- Transformational function reference

  Must be one of the following:

  - Transformational standard intrinsic function reference other than NULL, and all arguments are initialization expressions.
  - Either a transformational intrinsic function NULL reference defined by an expression other than an initialization expression, or arguments having assumed type parameters are not specified.
  - All arguments in the transformational function IEEE_SELECTED_REAL_KIND reference of the intrinsic module IEEE_ARITHMETIC are initialization expressions.

- Declaration query

  Must be one of the following:

  - An initialization expression query
  - An inherited item and must be a query other than items defined by an expression other than an unspecified expression or an initialization expression

- DO variable in an array constructor

  The corresponding DO type repeat shape specifications and DO type repeat strides must be initialization expressions.

- Initialization expression enclosed in parentheses

If an initialization expression references an object type parameter declared in the same declaration part or a query function related to an array shape specification, that type parameter or array shape specification must be declared in the leading part of the declaration part. Note that it does not matter if the declaration is to the left of the query function in the same statement, but it must not be in the same data element declaration.

The following intrinsic functions must not be specified in initialization expression.

ACOSH , ASINH , ATANH , BGE , BGT , BLE , BLT , DSHIFTL , DSHIFTR , HYPOT , LEADZ , MASKL , MASKR , MERGE_BITS , POPCNT , POPPAR , SHIFTA , SHIFTL , SHIFTR , STORAGE_SIZE , TRAILZ

The following intrinsic functions must not be specified a COMPLEX type argument in initialization expression.

ACOS , ASIN , ATAN , COSH , SINH , TAN , TANH

An intrinsic function ATAN must not be specified two arguments in initialization expression.

1.6.3 Intrinsic Operations

The intrinsic operators in descending order of precedence are:

Table 1.2 Intrinsic Operators

Operator	Represents	Operands	Precedence
**	exponentiation	two numeric	Highest
* and /	multiplication and division	two numeric	.
+ and -	unary addition and subtraction	one numeric	.
+ and -	binary addition and subtraction	two numeric	.
//	concatenation	two CHARACTER	.
.EQ. and == .NE. and /= .LT. and < .LE. and <=	equal to not equal to less than less than or equal to	two numeric or two CHARACTER ---------- two non-COMPLEX	.

Operator	Represents	Operands	Precedence
.GT. and > .GE. and >=	greater than greater than or equal to	numeric or two CHARACTER	
.NOT.	logical negation	one LOGICAL	.
.AND.	logical conjunction	two LOGICAL	.
.OR.	logical inclusive disjunction	two LOGICAL	.
.EQV. and .NEQV.	logical equivalence and non-equivalence	two LOGICAL	Lowest

An operation result of unary operation (unary +, unary -, and .NOT.) is of that type and has that kind type parameter. An operation result of relational operation is of type default LOGICAL. Otherwise, if an operation is performed on operands of the same type, the result is of that type and has the greater of the two kind type parameters. If an operation is performed on numeric operands of different types, the result is of the higher type, where COMPLEX is higher than REAL and REAL is higher than INTEGER, and has the greater of the two kind type parameters if operands are of type COMPLEX and of type REAL.

The result of a concatenation operation has a length that is the sum of the lengths of the operands.

1.6.4 Evaluation of Operands

Evaluation precedence of operands is not applied.

For example, on the following expression,

```
func(x) + func(y)
```

you should not require either `func(x)` or `func(y)` function reference is evaluated first.

The evaluation of a function reference shall neither affect nor be affected by the evaluation of any other entity within the statement.

For example, on the following expression,

```
func(i) .and. i==1
```

function reference `func(i)` shall not neither define nor undefine the variable `i`. However, execution of a function reference in the logical expression in an IF statement (see "2.264 IF Statement"), the mask expression in a WHERE statement (see "2.488 WHERE Statement"), or the subscripts and strides in a FORALL statement (see "2.178 FORALL Statement") is permitted to define variables in the statement that is conditionally executed.

The operands of an expression may not be evaluated when it is not necessary.

For example, on the following logical expression,

```
lop_1 .and. lfunc(i)
```

If `lop_1` has the value .FALSE., the result of expression is .FALSE. even if `lfunc(i)` is not evaluated.

1.7 Input/Output Statements

Input/output statements provide the means of transferring data form between external media or internal file and internal storage.

The input/output statements are the OPEN (see "2.354 OPEN Statement"), CLOSE (see "2.71 CLOSE Statement"), READ (see "2.389 READ Statement"), WRITE (see "2.489 WRITE Statement"), PRINT (see "2.369 PRINT Statement"), BACKSPACE (see "2.41 BACKSPACE Statement"), ENDFILE (see "2.134 ENDFILE Statement"), REWIND (see "2.398 REWIND Statement"), INQUIRE (see "2.272 INQUIRE Statement"), FLUSH (see "2.175 FLUSH Statement"), and WAIT (see "2.485 WAIT Statement") statements.

1.7.1 Fortran Records

A Fortran record is a sequence of values or a sequence of characters. One Fortran record does not always correspond to one physical record. The Fortran record types are as follows:

- Formatted

- Unformatted
- List-Directed
- Namelist
- Endfile
- Binary
- Stream

1.7.1.1 Formatted Fortran Records

Formatted Fortran records can consist of any character string. Formatted Fortran records are accessed by formatted sequential, formatted direct, an internal file input/output statements.

1.7.1.1.1 Formatted Sequential Records

Files controlled by formatted sequential input/output statements have an undefined length record format. One Fortran record corresponds to one logical record. The length of the undefined length record depends on the Fortran record to be processed. The maximum length may be assigned in OPEN statement RECL= specifier.

The line-feed character '\n' terminates the logical record. If the $ edit descriptor or \ edit descriptor is specified for the format of the formatted sequential output statement, the Fortran record does not include the line feed character '\n'.

1.7.1.1.2 Formatted Direct Records

File processed by formatted direct input/output statements have a fixed length record format. One Fortran record corresponds to one logical record. The length of the logical record shall be assigned in the OPEN statement RECL= specifier. If the Fortran record is shorter than the logical record. This fixed length record format is unique to Fortran.

1.7.1.1.3 Internal File Records

The fixed length records are processed by internal file input/output statements. The records are stored in internal files. Internal files that are a scalar character variable, character array element or substring contain only one Fortran record. The length of the Fortran record shall not exceed the size of the scalar character variable, character array element, or substring. When an internal file is a character array, the length of a Fortran record shall not exceed the size of an array element of the character array. If the length of the Fortran record is shorter than the size of the scalar character variable, character array element or substring, the remaining part is padded with blanks. For input, the length of the Fortran record shall be equal.

1.7.1.2 Unformatted Fortran Records

An unformatted Fortran record consists of a string of values (character or noncharacter data, and a string can be empty). The length depends on the input/output source program statement (zero length can be used). Both unformatted sequential and unformatted direct input/output statements can use unformatted Fortran records.

1.7.1.2.1 Unformatted Sequential Records

Files processed using unformatted sequential input/output statements have a variable length record format. One Fortran record corresponds to one logical record. The length of the variable length record depends on the length of the Fortran record. The length of the Fortran record includes 4 bytes added to the beginning and end of the logical record. The maximum length may be assigned in the OPEN statement RECL= specifier. The beginning area is used when an unformatted sequential statement is executed. The end area is used when a BACKSPACE statement is executed. This variable length record format is unique to Fortran.

1.7.1.2.2 Unformatted Direct Records

Files processed by unformatted direct input/output statements have a fixed length record format. One Fortran record can correspond to more than one logical record (see "1.7.1.1.2 Formatted Direct Records").

The record length shall be assigned in the OPEN statement RECL= specifier. One Fortran record can consist of more than one logical record. If the Fortran record terminates within a logical record, the remaining part is padded with binary zeros. If the length of the Fortran record exceeds the logical record, the remaining data goes into the next record. This fixed length record format is unique to Fortran.

1.7.1.3 List-Directed Fortran Records

List-directed input/output statements, print statement, or internal file input/output statements are used to process list-directed Fortran records. List-Directed Fortran records consist of data items and value separations. The data items are input/output character strings. The length of the Fortran record depends on the number and type of items. Data items from the beginning of a logical record up to the end of the input are handled as one Fortran record. Files processed using list-directed input/output statements have an undefined length record format or a fixed length record format (see "1.7.1.1.1 Formatted Sequential Records" or "1.7.1.1.3 Internal File Records").

1.7.1.4 Namelist Fortran Records

Namelist input/output statements and internal input/output statements access namelist Fortran records. Namelist Fortran records consist of data items (character strings specified by a namelist name) from the &namelist name up to / or &end. The correspondence between namelist Fortran records and logical records is the same as for Fortran records handled using list-directed input/output statements. File accessed by namelist input/output statements have an undefined length record format (see "1.7.1.1.1 Formatted Sequential Records" or "1.7.1.1.3 Internal File Records").

1.7.1.5 Endfile Records

An endfile record shall be the last record of a sequential file. Endfile records do not have a length attribute. The ENDFILE statement writes endfile records in sequential files. After at least one WRITE statements is executed, endfile records are output under the following conditions:

- A REWIND statement is executed.
- A BACKSPACE statement is executed.
- A CLOSE statement is executed.

1.7.1.6 Binary Fortran Records

A binary Fortran record consists of a string of values (character or noncharacter data). The length of the binary Fortran record depends on the input/output list (it may be zero length). One Fortran record corresponds to one logical record. An unformatted sequential access or an unformatted direct access input/output statement may access binary Fortran records.

A binary record is a fixed length record format.

1.7.1.7 Stream Fortran Records

A stream Fortran record is comprised of storage units that can be uniquely identified by positive integers. Fortran records can be handled by stream access input/output statements.

1.7.2 Files

Input/Output statements process internal and external files. Internal files are stored in main memory.

Internal files consist of scalar character variables, character array elements, character arrays, or substrings. Internal input/output statements may read or write internal files. External files are stored on external devices and are classified as sequential or direct access files. Sequential access files are processed by Fortran sequential access methods. Direct access files are processed by the Fortran direct access methods. Fortran programs process the following types of files:

- Standard input files (stdin)
- Standard output files (stdout)
- Standard error output files (stderr)
- Regular files

Both sequential and direct access methods can be used to access regular files. Only the sequential access method can be used to access standard files.

1.7.2.1 Old and New Files

New files may be created when an executable program is started or at any time during its execution.

Old and new files may be accessed when connected to a unit number specified in an input/output statement (see "1.7.3.1 Unit Numbers and File Connection").

1.7.2.2 File Position

Certain input/output statements affect the position within an external file. Prior to execution of a data transfer statement, a direct file is positioned at the beginning of the record indicated by the record specifier REC= in the data transfer statement. By default, a sequential file is positioned after the last record read or written. However, if non-advancing input/output is specified using the ADVANCE= specifier, it is possible to read or write partial records and to read variable-length records and be notified of their length.

An ENDFILE statement writes an endfile record after the last record read or written and positions the file after the endfile record. A REWIND statement positions the file at its initial point. A BACKSPACE statement moves the file position back one record.

If an error condition occurs, the position of the file is indeterminate.

If there is no error, and an endfile record is read or written, the file is positioned after the endfile record. The file shall be repositioned with a REWIND or BACKSPACE statement before it is read from or written to again.

1.7.2.3 Internal Files

An internal file is always a formatted sequential file and consists of a single CHARACTER variable. If the CHARACTER variable is array-valued, each element of the array is treated as a record in the file. The CHARACTER variable shall not be an array section with a vector subscript.

1.7.3 File Connection

A unit provides a means for referring to a file.

A unit is one of the following:

- external file unit
- *
- internal file unit

An external file unit is a scalar INTEGER expression. An external unit is used to refer to an external file and is specified by an external file unit or an asterisk '*'. An internal file unit is a default CHARACTER variable. An internal unit is used to refer to an internal file and is specified by an internal file unit.

1.7.3.1 Unit Numbers and File Connection

A file and unit shall be connected to execute a data transfer or file positioning input/output statement. The CLOSE statement is used to terminate the connection of a unit and to an external file.

A unit shall not be connected to more than one file at the same time, and file shall not be connected to more than one unit at the same time. After an external unit has been disassociated by the execution of a CLOSE statement, it may be connected again within the same program to the same file or to a different file. After an external file has been disconnected by the execution of a CLOSE statement, it may be connected again within the same program to the same unit or to a different unit.

1.7.3.2 Preconnected Units

Preconnection means that the unit is connected to a file at the beginning of execution of the program and therefore it may be specified input/output statements without the prior execution of an OPEN statement. Units 0, 5, and 6 are preconnected to standard error output file, standard input file, and standard output file respectively.

Unit * is always connected to the standard input and standard output devices. A READ statement that does not contain UNIT= specifier and PRINT statement specify the unit *.

1.8 Input/Output Editing

A format used in conjunction with an input/output statement provides information that directs the editing between the internal representation of data and the characters of a sequence of formatted records.

A format may be provided explicitly format specification, may be list-direct formatting (see "1.8.2 List-Directed Formatting"), and may be namelist formatting (see "1.8.3 Namelist Formatting"). Explicit format specification may be given in a FORMAT statement or in a character expression for FMT= specifier.

1.8.1 Format Specification

The syntax for a format specification is

```
( [ format-item-list ] )
```

The comma used to separate *format-items* in a *format-item-list* may be omitted in the following case:

- Between a P edit descriptor and an immediately following F, E, EN, ES, D, G, or Q*w.d* edit descriptor.
- Before a slash edit descriptor when the optional repeat specification is not present.
- After a slash edit descriptor.
- Before or after a colon edit descriptor.

format-item is:

```
[ r ] data-edit-descriptor          or
control-edit-descriptor             or
char-string-edit-descriptor         or
[ r ] ( format-item-list )          or
[ r ] Q
```

r shall be a positive INTEGER literal constant that shall not be specified a kind parameter, shall have the value between 1 and 32767. *r* is called a repeat specification.

data-edit-descriptor is:

```
I[w[.m]]                              or
B[w[.m]]                              or
O[w[.m]]                              or
Z[w[.m]]                              or
F[w[.d]]                              or
E[w[.d[Ee]]]                          or
E[w[.d[De]]]                          or
ENw.d[Ee]                             or
ESw.d[Ee]                             or
G [w[.d[Ee]]]                         or
L [w]                                 or
A [w]                                 or
D [w[.d]]                             or
DT[char-literal-constant][(v-list)]   or
Qw.d
```

char-literal-constant is a CHARACTER literal constant.

v is a signed INTEGER literal constant.

w, *m*, *d*, and *e* shall be a positive INTEGER literal constant that shall not be specified a kind parameter, shall have the value between 0 and 255 except A edit descriptor. For A edit descriptor, *w* shall be a positive INTEGER literal constant that shall not be specified a kind parameter, shall have the value between 0 and 65000.

control-edit-descriptor is:

```
position-edit-descriptor            or
[r] /                               or
:                                   or
sign-edit-descriptor                or
kP                                  or
blank-interp-edit-descriptor        or
round-edit-descriptor               or
decimal-edit-descriptor             or
```

```
$                                   or
\                                   or
[n]R
```

k shall be a signed INTEGER literal constant that shall not be specified a kind parameter, shall have the value between -127 and 127.

n shall be a positive INTEGER literal constant that shall not be specified a kind parameter, shall have the value between 2 and 36.

position-edit-descriptor is:

```
Tn          or
TLn         or
TRn         or
nX
```

n shall be a positive INTEGER literal constant that shall not be specified a kind parameter, shall have the value between 1 and 32760.

sign-edit-descriptor is:

```
S           or
SP          or
SS
```

blank-interp-edit-descriptor is:

```
BN          or
BZ
```

char-string-edit-descriptor is CHARACTER literal constant enclosed apostrophes or quotation marks that shall not be specified a kind parameter, or H edit descriptor (deleted feature).

round-edit-descriptor is the following round edit descriptor:

```
RU          or
RD          or
RZ          or
RN          or
RC          or
RP
```

decimal-edit-descriptor is the following decimal edit descriptor:

```
DC          or
DP
```

1.8.1.1 Format Control

Except for a format item preceded by a repeat specification *r*, a format specification is interpreted from left to right. A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the format item but without the repeat specification and separated by commas.

To each data edit descriptor interpreted in a format specification, there corresponds one effective item specified by the input/output list, except that an input/output list item of type complex requires the interpretation of two F, E, EN, ES, D, G, or Q *w.d* edit descriptors. For each control edit descriptor or character edit descriptor, there is no corresponding item specified by the input/output list.

If format control encounters the rightmost parenthesis of a complete format specification and another effective input/output list item is specified, the file is positioned in a manner identical to the way it is positioned when a slash edit descriptor is processed (see "1.8.1.3.2 Slash Editing"). Format control then reverts to the beginning of the format item terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If format control reverts to a parenthesis that is preceded by a repeat specification *r*, the repeat specification *r* is reused.

1.8.1.2 Data Edit Descriptors

Data edit descriptors control cause the conversion of data to or from its internal representation. All data edit descriptors may be used to specify the input/output of any intrinsic type data.

1.8.1.2.1 Numeric Editing

The I, B, O, Z, F, E, EN, ES, D, G, and Q*w.d* edit descriptors can be used to specify the input/output of INTEGER, REAL, and COMPLEX data. The following general rules apply:

- On input, leading blanks are not significant.
- On input, a decimal point appearing in the input field overrides the position of an edit descriptor that specifies the decimal point location.
- On input, the field width cannot be zero.
- On output, the representation is right-justified in the field.
- On output, if the number of characters produced exceeds the field width the entire field is filled with asterisks.
- On output, the minus sign is not output when the internal data value is zero.

1.8.1.2.2 INTEGER Editing

The I*w*, I*w.m*, B*w*, B*w.m*, O*w*, O*w.m*, Z*w*, and Z*w.m* edit descriptors indicate that the field to be edited occupies *w* positions, except when *w* is zero. When *w* is zero, a suitable width will be used to show all digits without any padding blanks. On input, *w* shall not be zero. When *w* is not specified, the field width is selected as the following table.

Edit descriptor	Input/output item type			
	One-byte INTEGER	Two-byte INTEGER	Four-byte INTEGER	Eight-byte INTEGER
I edit descriptor	4	6	11	20
B edit descriptor	9	17	33	65
O edit descriptor	4	7	12	23
Z edit descriptor	3	5	9	17

When *m* is specified for output editing, if necessary, sufficient leading zeros are included to achieve the minimum of *m* digits. The value of *m* shall not exceed the value of *w*, except *w* is zero. If *m* is zero and the value of the internal datum is zero, the output field consists of only blank characters. When *m* and *w* are both zero, and the value of the internal datum is zero, one blank character is produced.

On input, *m* has no effect.

1.8.1.2.3 REAL and COMPLEX Editing

The F*w.d*, E*w.d*, D*w.d*, E*w.d*E*e*, E*w.d*D*e*, EN, ES, and Q*w.d* edit descriptors indicate the manner of editing of REAL and COMPLEX data. The editing of a datum of complex data type is specified by two edit descriptors (see "1.8.1.2.4 COMPLEX Editing").

F, E, D, EN, ES, and Q*w.d* editing are identical on input. The *w* indicates the width of the field; the *d* indicates the number of digits in the fractional part. The *w* shall not be zero. When *w* is not specified, the field width is selected as the following table.

Edit descriptor	Input/output item type		
	Single precision REAL, COMPLEX	Double precision REAL, COMPLEX	Quadruple precision REAL, COMPLEX
F edit descriptor	15	22	43
E edit descriptor	15	22	43
D edit descriptor	15	22	43
Q*w.d* edit descriptor	15	22	43

The *d* has no effect on input if the input field contains a decimal point. If the decimal point is omitted, the rightmost *d* digits are interpreted as the fractional part. The *e* has no effect in input. An exponent can be included in one of the following forms:

- An explicitly signed INTEGER constant.
- E, D, or Q followed by an optionally signed INTEGER constant.

F editing, the output field consists of zero or more blanks followed by a minus sign or an optional plus sign (see "1.8.1.3.4 S, SP, and SS Editing"), followed by one or more digits that contain a decimal point and represent the magnitude. The field is modified by the established scale factor (see "1.8.1.3.5 P Editing") and is rounded to *d* decimal digits. If *w* is zero then a suitable width will be used to show all digits and sign without any padding blanks.

For E, D, and Q *w.d* edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of *d* digits, unless a scale factor greater than one is in effect, and the exponent part consists of *e* digits. The output field consists of the following, in order:

1. zero or more blanks
2. a minus or an optional plus sign (see "1.8.1.3.4 S, SP, and SS Editing")
3. a zero (depending on scale factor, see "1.8.1.3.5 P Editing")
4. Decimal point

   If the decimal point editing mode is POINT, the decimal point symbol is the period '.', but if the decimal point editing mode is COMMA, the decimal point symbol is the comma ','.
5. the *d* most significant digits, rounded
6. an E, a D, or a Q if exponent is two or less digits, or E *w.d*E *e*, E *w.d*D *e* editing
7. a plus or a minus sign
8. an exponent of *e* digits, if the extended E *w.d*E *e* form is used, and two digits otherwise.

The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit descriptor forms E *w.d*, D *w.d*, and Q *w.d* shall not be used if |*exp*| > 999.

For E, D, and Q *w.d* editing, the scale factor *k* controls the position of the decimal point (see "1.8.1.3.5 P Editing"). If -*d* < *k* <= 0, the output field contains exactly |*k*| leading zeros and *d* - |*k*| significant digits after the decimal point. If 0 < *k* < *d*+2 , the output field contains exactly *k* significant digits to the left of the decimal point and *d*-*k*+1 significant digits to the right of the decimal point. Other values of *k* are not permitted.

The EN edit descriptor produces an output field in the form of a real number in engineering notation such that the decimal exponent is divisible by three and the absolute value of the significand is greater than or equal to 1 and less than 1000, except when the output value is zero. The scale factor has no effect on output.

The forms of the edit descriptor are EN *w.d* and EN *w.d*E *e* indicating that the external field occupies *w* positions, the fractional part of which consists of *d* digits and the exponent part *e* digits.

The ES edit descriptor produces an output field in the form of a real number in scientific notation such that the absolute value of the significand is greater than or equal to 1 and less than 10, except when the output value is zero. The scale factor has no effect on output.

The forms of the edit descriptor are ES *w.d* and ES *w.d*E *e* indicating that the external field occupies *w* positions, the fractional part of which consists of *d* digits and the exponent part *e* digits.

The IEEE exception specification field is any field in between two optional blanks.

1. The character string "INF" or "INFINITY" preceded by an optional sign.
2. The character string "NAN" preceded by an optional sign. An alphanumeric of zero or greater enclosed in parentheses can follow this.

The value regulated by Format 1. is IEEE infinity. The value regulated by Format 2. is IEEE NaN. When the non-blank input field is "NAN" or "NAN()", the NaN value is used as an exception unknown NaN.

When the internal value is IEEE infinity, the output field consists of a blank (if required) a minus sign if followed by minus infinity, or an optional plus sign at other times, followed by the character string "Inf" or "Infinity". The output field is right justified. If w is less than 3, the field is padded with asterisks, and if w is greater than or equal to 3 and less than 8, "Inf" if generated.

When the internal value is IEEE NaN, the output field consists of a blank (if required) followed by the character string "NaN", followed by one or more optional alphanumerics with w of five characters or less enclosed in quotation marks. It is right justified. If w is less than 3, the field is padded with asterisks.

When the internal value is neither IEEE infinity nor IEEE NaN, the output field consists of a blank (if required) a minus sign if followed by a minus internal value, or an optional plus sign at other times, followed by a numeric string containing one decimal point symbol. This

numeric string is the absolute value of the internal value corrected by the logical shift number and is rounded to d digits after the decimal point symbol.

1.8.1.2.4 COMPLEX Editing

COMPLEX editing is accomplished by using two REAL edit descriptors. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors can be different. Control and character string edit descriptors (see "1.8.1.3 Control Edit Descriptors" and "1.8.1.4 Character String Edit Descriptors") may be processed between the edit descriptor for the real part and the edit descriptor for the imaginary part.

1.8.1.2.5 LOGICAL Editing

The L*w* edit descriptor indicates that the field occupies *w* positions.

The input field consists of optional blanks, optionally followed by a period '.', followed by a T for true or F for false. The T or F can be a lowercase letter: t or f. The T or F can be followed by additional characters in the field. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

The output field consists of *w*-1 blanks followed by a T or F, depending on whether the value of the internal data object is true or false, respectively.

When *w* is not specified, the field width is selected as the following table.

Edit descriptor	Input/output item type			
	One-byte LOGICAL	Two-byte LOGICAL	Four-byte LOGICAL	Eight-byte LOGICAL
L edit descriptor	2	2	2	2

1.8.1.2.6 CHARACTER Editing

The A*w* edit descriptor is used with an input/output list item of type CHARACTER.

If a field width *w* is specified with the A edit descriptor, the field consists of *w* characters. If a field width *w* is not specified with the A edit descriptor, the number of characters in the field is the length of the corresponding list item.

Let *len* be the length of the list item. On input, if *w* is greater than or equal to *len*, the rightmost *len* characters will be taken from the field; if *w* is less than *len*, the *w* characters are left-justified and padded with *len*-*w* trailing blanks.

On output, the list item is padded with leading blanks if *w* is greater than *len*. If *w* is less than or equal to *len*, the output field consists of the leftmost *w* characters of the list item.

1.8.1.2.7 G Editing

The G*w*, G*w.d*, and G*w.d*E*e* edit descriptors can be used with an input/output list item of any intrinsic type.

These edit descriptors indicate that the external field occupies *w* positions, the fractional part of which consists of a maximum of *d* digits and the exponent part *e* digits. *d* and *e* have no effect when used with INTEGER, LOGICAL, or CHARACTER data.

When *w* is not specified, the field width is selected as the following table.

Edit descriptor	Input/output item type										
	One-byte INTEGER	Two-byte INTEGER	Four-byte INTEGER	Eight-byte INTEGER	Single precision REAL,COMPLEX	Double precision REAL,COMPLEX	Quadruple precision REAL,COMPLEX	One-byte LOGICAL	Two-byte LOGICAL	Four-byte LOGICAL	Eight-byte LOGICAL
G edit descriptor	4	6	11	20	15	22	43	2	2	2	2

With INTEGER data, the G*w*, G*w.d*, and G*w.d*E*e* edit descriptors follow the rules for the I*w* edit descriptor.

With REAL or COMPLEX data, the form and interpretation of the input field is the same as for F*w* edit descriptor. Of the output field, the absolute-value to be printed is in range of the F edit descriptor, then the output is as for the F edit descriptor; otherwise, editing is as for the E edit descriptor. If an absolute-value of a data is in range of the F edit descriptor, then the output is as for the scale factor *k* (see "1.8.1.3.5 P Editing") has no effect. The G*w* edit descriptor is the same as for the G*w*.0 was specified.

With LOGICAL data, the G*w*, G*w.d*, and G*w.d*E*e* edit descriptors follow the L*w* edit descriptor rules.

With CHARACTER data, the G*w*, G*w.d*, and G*w.d*E*e* edit descriptors follow the A*w* edit descriptor rules.

1.8.1.3 Control Edit Descriptors

A control edit descriptor does not cause the transfer of data or the conversion of data to or from internal representation, but may affect the conversions performed by subsequent data edit descriptors.

1.8.1.3.1 Position Editing

The T*n*, TL*n*, TR*n*, and *n*X edit descriptors control the character position in the current record to or from which the next character will be transferred. The new position can be in either direction from the current position. This makes possible the input of the same record twice, possibly with different editing. It also makes skipping characters in a record possible.

The T*n* edit descriptor tabs to character position *n* from the beginning of the record. The TL*n* and TR*n* edit descriptors tab *n* characters left or right, respectively, from the current position. The *n*X edit descriptor tabs *n* characters right from the current position, the same as TR*n*.

On output, a T, TL, TR, or X edit descriptor does not by itself affect the length of the record. If the position is changed to beyond the length of the current record, the next data transfer to or from the record causes the insertion of blanks in the character positions not previously filled.

1.8.1.3.2 Slash Editing

The slash edit descriptor terminates data transfer to or from the current record. The file position advances to the beginning of the next record. On output to a file connected for sequential access, a new record is written and the new record becomes the last record in the file.

1.8.1.3.3 Colon Editing

The colon edit descriptor terminates format control if there are no more items in the input/output list. The colon edit descriptor has no effect if there are more items in the input/output list.

1.8.1.3.4 S, SP, and SS Editing

The S, SP, and SS edit descriptors control whether an optional plus is to be transmitted in subsequent numeric output fields. SP causes the optional plus to be transmitted. SS causes it not to be transmitted. S returns optional pluses to the default (no pluses).

1.8.1.3.5 P Editing

The *k*P edit descriptor sets the value of the scale factor to *k*. The scale factor affects the F, E, EN, ES, D, G, or Q*w.d* editing of subsequent numeric quantities as follows:

- On input (provided that no exponent exists in the field) the scale factor causes the externally represented number to be equal to the internally represented number multiplied by 10^{k}. The scale factor has no effect if there is an exponent in the field.
- On output, with F editing, the scale factor effect is that the externally represented number equals the internally represented number times 10^{k}.
- On output, with E, D, and Q*w.d* editing, the significand part of the quantity to be produced is multiplied by 10^{k} and the exponent is reduced by *k*.
- On output, with G editing, the effect of the scale factor is suspended unless the magnitude of the data object to be edited is outside the range that permits the use of F editing. If the use of E editing is required, the scale factor has the same effect as with E output editing.
- On output, with EN and ES editing, the scale factor has no effect.

1.8.1.3.6 BN and BZ Editing

The BN and BZ edit descriptors are used to specify the interpretation, by numeric edit descriptors, of non-leading blanks in subsequent numeric input fields. If a BN edit descriptor is encountered in a format, blanks in subsequent numeric input fields are ignored. If a BZ edit descriptor is encountered, blanks in subsequent numeric input fields are treated as zeros.

1.8.1.3.7 $ Editing

A $ edit descriptor indicates that the next item is written in the same record for the formatted sequential access output statement without the current record being terminated (that is, without writing the line-feed character).

1.8.1.3.8 \ Editing

A \ edit descriptor is the same as $ edit descriptor (see "1.8.1.3.7 $ Editing").

1.8.1.3.9 R Editing

An R edit descriptor specifies a radix which may be other than 10.

If n is not specified, the default is 10. If n is specified, n must be an unsigned INTEGER constant and assigns the value between 2 and 36.

For example,

```
read ("101","(5r,i3)") n
```

converts 101 in base (radix) 5 to decimal 26 and assigns the value 26 to n.

1.8.1.3.10 RU, RD, RZ, RN, RC, and RP Editing

The round edit descriptors temporarily change the connection input/output rounding mode.

The RU, RD, RZ, RN, RC, and RP round edit descriptors set the input/output rounding modes corresponding to the respective ROUND= specifier values UP, DOWN, ZERO, NEAREST, COMPATIBLE, and PROCESSOR_DEFINED. The input/output rounding mode affects the conversion of real number and complex number values in formatted input/output. Only D, E, EN, ES, F, and G editing is affected.

If the input/output rounding mode is UP, the value in the conversion results is the minimum value that can be expressed equal to or greater than the original value. If the input/output rounding mode is DOWN, the value in the conversion results is the maximum value that can be expressed equal to or less than the original value. If the input/output rounding mode is ZERO, the value in the conversion results is an absolute value that is less than and closest to the original value. If the input/output rounding mode is NEAREST, the value in the conversion results is the nearest value that can be expressed and, if the rounding result is midway between two values that can be expressed, it is an even number value. If the input/output rounding mode is COMPATIBLE, the value in the conversion results is the nearest value that can be expressed and, if the rounding result is midway between two values that can be expressed, it is the value furthest from zero. If the input/output rounding mode is PROCESSOR_DEFINED, it is assumed that the connection input/output rounding mode is omitted. In this case, the result depends on the floating-point rounding mode.

1.8.1.3.11 User-defined Derived-type Editing

Instead of implicit input/output editing, DT editing descriptors can use a procedure prepared by users for processing derived type list items.

1.8.1.3.12 DC and DP Editing

The decimal point descriptor temporarily changes the connection decimal point editing mode.

The DC and DP editing descriptors set the decimal point editing modes corresponding respectively to the 'COMMA' and 'POINT' for DECIMAL= specifier values.

The decimal point editing mode is used to control the decimal point symbol expression when converting real number and complex number values in formatted input/output. The decimal point symbol is the character that separates the integer part and the fraction part in real number decimal expressions in internal and external files.

The decimal point editing mode affects only D, E, EN, ES, F, and G editing. When the decimal point editing mode is POINT, the decimal point symbol is the period. When the decimal point editing mode is COMMA, the decimal point symbol is the comma.

When the decimal point editing mode is COMMA for list input/output, a semicolon is used as the value delimiter instead of the comma.

1.8.1.4 Character String Edit Descriptors

A character string edit descriptors cause literal CHARACTER data to be output. It shall not be used for input.

The CHARACTER string edit descriptor causes characters to be output from a string, including blanks. Enclosing characters are either apostrophes or quotation marks, or H edit descriptor.

For a CHARACTER string edit descriptor, the width of the field is the number of characters contained in, but not including, the delimiting characters. Within the field, two consecutive delimiting characters (apostrophes, if apostrophes are the delimiters; quotation marks, if quotation marks are the delimiters) are counted as a single character. Thus an apostrophe or quotation mark character can be output as part of a CHARACTER string edit descriptor delimited by the same character.

1.8.1.4.1 H Editing (deleted feature)

An H edit descriptor is the following syntax:

*c*H*char*[*char*]...

Where:

c shall be a positive INTEGER that shall not be specified a kind parameter, shall have the value 1 to 65000.

char shall be of default CHARACTER type.

c specifies the number of characters following the H. The width of the field is *c*.

1.8.1.5 Q Edit Descriptor

A Q edit descriptor returns the remaining number of characters in the current input record. The corresponding input item shall be INTEGER type.

1.8.2 List-Directed Formatting

List-directed formatting is indicated when an input/output statement uses an asterisk instead of an explicit format.

For example,

```
read *,a
print *,x,y,z
write (unit=*,fmt=*) i,j,k
```

all use list-directed formatting.

1.8.2.1 List-Directed Input

List-directed records consist of a sequence of values and value separators.

Values are either null or any of the following forms:

```
c       or
r*c     or
r*
```

Where:

c is a literal constant that shall not be specified a kind parameter or a non-delimited CHARACTER string.

r is a positive INTEGER literal constant with no kind type parameter specified.

*r***c* is equivalent to *r* successive instances of *c*.

*r** is equivalent to *r* successive instances of null.

Separators are either commas, slashes, or line feed characters with optional preceding or following blanks; or one or more blanks or horizontal tab character between two non-blank values. A slash separator causes termination of the input statement after transfer of the previous value.

When the decimal point editing mode is POINT, the delimiter is the comma, and when the decimal point editing mode is COMMA, the delimiter is the semicolon.

When the character string enclosure symbol is omitted, the character string ends at the first blank space, comma (if the decimal point editing mode is POINT), semicolon (if the decimal point editing mode is COMMA), forward slash, or at the end of the record. Two consecutive apostrophes or quotation marks are not used partway through data.

Editing occurs based on the type of the list item as explained below. On input the following formatting applies:

Type	Editing
INTEGER	I
REAL	F
COMPLEX	As for COMPLEX literal constant
LOGICAL	L
CHARACTER	As for CHARACTER string. CHARACTER string can be continued from one record to the next. Delimiting apostrophes or quotation marks are not required if the CHARACTER string does not cross a record boundary and does not contain value separators or CHARACTER string delimiters.

1.8.2.2 List-Directed Output

When the decimal point editing mode is POINT, the delimiter is the comma, and when the decimal point editing mode is COMMA, the delimiter is the semicolon.

For list-directed output the following formatting applies:

Type	Editing
INTEGER	I*w*
REAL	F*w* or E*w.d*E*e*
COMPLEX	(F*w* or E*w.d*E*e*, F*w* or E*w.d*E*e*)
LOGICAL	T for value true and F for value false
CHARACTER	As CHARACTER string, except as overridden by the DELIM= specifier

1.8.3 Namelist Formatting

Namelist formatting is indicated by an input/output statement with an NML= specifier.

Namelist input and output 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 separations, and
5. a slash or '&end' to terminate the namelist input.

A *name-value* subsequence is a data object or subobject previously declared in a NAMELIST statement to be part of the namelist group, followed by an equals, followed by one or more values and value separators. Formatting for namelist records is the same as for list-directed records (see "1.8.2 List-Directed Formatting").

Except within a character literal constant, a '!' character after a value separator or in the first nonblank position of a namelist input record initiates a comment. The comment extends to the end of the current input record.

If an effective item is the COMPLEX type, the input format consists of a left parenthesis, followed by an ordered set of numeric value input fields delimited by commas (if the decimal point editing mode is POINT) or semicolons (if the decimal point editing mode is COMMA), followed by the right parenthesis. When the decimal point editing mode is POINT, the delimiter is the comma, and when the decimal point editing mode is COMMA, the delimiter is the semicolon.

For example,

```
integer :: i,j(10)
real :: n(5)
namelist /my_namelist/ i,j,n
read(*,nml=my_namelist)
```

if the input records that correspond to above program are

```
&my_namelist i=5, n(3)=4.5,
j(1:4)=4*0/
```

then `5` is stored in `i`, `4.5` in `n(3)`, and `0` in elements 1 through 4 of `j`.

1.9 Statements

Each Fortran statement is classified as either an executable statement or a nonexecutable statement. An executable statement performs or controls actions, and determines the computational behavior of the program. A nonexecutable statement is used to configure the program environment. The nonexecutable statements are all those not classified as executable.

A brief description of each statement follows. For complete syntax and rules, see "Chapter 2 Alphabetical Reference".

1.9.1 Executable Statements

ALLOCATE

The ALLOCATE statement dynamically creates pointer targets and allocatable variables (see "2.16 ALLOCATE Statement").

Arithmetic IF (obsolescent feature)

Execution of arithmetic IF statement causes evaluation of an expression followed by a transfer of control (see "2.20 Arithmetic IF Statement (obsolescent feature)").

ASSIGN (deleted feature)

Assigns a statement label to a scalar INTEGER variable (see "2.25 ASSIGN Statement (deleted feature)").

Assigned GO TO (deleted feature)

The assigned GO TO statement causes a transfer of control to the branch target statement indicated by a variable that was assigned a statement label in an ASSIGN statement (see "2.26 Assigned GO TO Statement (deleted feature)").

Assignment

Assigns the value of the expression on the right side of the equal sign to the variable on the left side of the equal sign (see "2.27 Assignment Statement").

ASSOCIATE

The ASSOCIATE statement associates an expression or variable with a named element (see "2.28 ASSOCIATE Construct").

BACKSPACE

The BACKSPACE statement positions the file before the current record, if there is a current record, otherwise before the preceding record (see "2.41 BACKSPACE Statement").

BLOCK construct

The BLOCK construct is an executable construct that may include declarations (see "2.50 BLOCK Construct").

CALL

The CALL statement invokes a subroutine and passes to it a list of arguments (see "2.55 CALL Statement").

CASE

The CASE statement shall be specified in a CASE construct, and specifies the condition to execute the following block (see "2.56 CASE Construct" and "2.57 CASE Statement").

CLOSE

The CLOSE statement terminates the connection of a specified input/output unit to an external file (see "2.71 CLOSE Statement").

Computed GO TO (obsolescent feature)

The computed GO TO statement causes transfer of control to one of a list of labeled statements or immediately following statement (see "2.79 Computed GO TO Statement (obsolescent feature)").

CONTINUE

Execution of a CONTINUE statement has no effect (see "2.83 CONTINUE Statement").

CRITICAL construct

The CRITICAL construct prevents more than one image from executing a block at a same time (see "2.96 CRITICAL Construct").

CYCLE

The CYCLE statement curtails the execution of a single iteration of a DO loop (see "2.99 CYCLE Statement").

DEALLOCATE

The DEALLOCATE statement deallocates allocatable variables and pointer targets and disassociates pointers (see "2.110 DEALLOCATE Statement").

DO

The DO statement begins a DO construct. A DO construct specifies the repeated execution (loop) of a sequence of executable statements or constructs (see "2.114 DO Construct").

ELSE

The ELSE statement shall be specified in an IF construct, and controls conditional execution of a following block where all previous IF expressions are false (see "2.263 IF Construct" and "2.124 ELSE Statement").

ELSE IF

The ELSE IF statement shall be specified in an IF construct, and specifies the condition to execute the following block where all previous IF expressions are false (see "2.263 IF Construct" and "2.125 ELSE IF Statement").

ELSEWHERE

The ELSEWHERE statement shall be specified in a WHERE construct, and controls conditional execution for elements of an array for which all previous WHERE construct's mask expression is false (see "2.486 WHERE Construct" and "2.126 ELSEWHERE Statement").

END

The END statement ends a program unit or subprogram (see "2.127 END Statement").

END ASSOCIATE

The END ASSOCIATE statement ends an ASSOCIATE construct (see "2.28 ASSOCIATE Construct" and "2.128 END ASSOCIATE Statement").

END BLOCK

The END BLOCK statement ends a BLOCK construct (see "2.50 BLOCK Construct" and "2.129 END BLOCK Statement").

END CRITICAL

The END CRITICAL statement ends a CRITICAL construct (see "2.96 CRITICAL Construct" and "2.131 END CRITICAL Statement").

END DO

The END DO statement ends a DO construct (see "2.114 DO Construct" and "2.132 END DO Statement").

ENDFILE

The ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record, which becomes the last record of the file (see "2.134 ENDFILE Statement").

END FORALL

The END FORALL statement ends a FORALL construct (see "2.176 FORALL Construct" and "2.135 END FORALL Statement").

END FUNCTION

The END FUNCTION statement ends a function subprogram (see "2.136 END FUNCTION Statement" and "1.12.1 Function Subprogram").

END IF

The END IF statement ends an IF construct (see "2.263 IF Construct" and "2.137 END IF Statement").

END PROGRAM

The END PROGRAM statement ends a main program (see "2.141 END PROGRAM Statement" and "1.11.1 Main Program").

END SELECT

The END SELECT statement ends a CASE construct or a SELECT TYPE construct (see "2.56 CASE Construct", "2.142 END SELECT Statement" and "2.413 SELECT TYPE Construct").

END SUBROUTINE

The END SUBROUTINE statement ends a subroutine subprogram (see "2.144 END SUBROUTINE Statement" and "1.12.2 Subroutine Subprogram").

END WHERE

The END WHERE statement ends a WHERE construct (see "2.486 WHERE Construct" and "2.147 END WHERE Statement").

ERROR STOP

The ERROR STOP statement terminates execution of the program as an error (see "2.155 ERROR STOP Statement").

EXIT

The EXIT statement terminates a DO loop (see "2.162 EXIT Statement").

FLUSH

The FLUSH statement is effective items as follows when executing it (see "2.175 FLUSH Statement").

- A data written in an external file can be used by other processing.
- A data in an external file excluding Fortran can be used by a READ statement.

FORALL

The FORALL statement controls a single assignment or pointer assignment statement (see "2.178 FORALL Statement").

FORALL construct

The FORALL construct statement begins a FORALL construct. The FORALL construct controls multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements (see "2.176 FORALL Construct").

GO TO

The GO TO statement transfers control to a statement identified by a label (see "2.210 GO TO Statement").

IF

The IF statement controls whether or not a single executable statement is executed (see "2.264 IF Statement").

IF THEN

The IF THEN statement begins an IF construct. The IF construct selects for execution no more than one of its constituent blocks (see "2.263 IF Construct").

INQUIRE

The INQUIRE statement inquires about properties of named file or of the connection to a unit (see "2.272 INQUIRE Statement").

LOCK

The LOCK statement locks lock variables (see "2.311 LOCK Statement").

NULLIFY

The NULLIFY statement disassociates pointers (see "2.351 NULLIFY Statement").

OPEN

The OPEN statement connects or reconnects an external file and an input/output unit (see "2.354 OPEN Statement").

PAUSE (deleted feature)

The PAUSE statement temporarily suspends execution of the program (see "2.359 PAUSE Statement (deleted feature)").

Pointer Assignment

The pointer assignment statement associates a pointer with a target (see "2.363 Pointer Assignment Statement").

PRINT

The PRINT statement transfers values from an output list and format specification to a file (see "2.369 PRINT Statement").

READ

The READ statement transfers values from a file to the entities specified in an input list or a namelist group (see "2.389 READ Statement").

RETURN

The RETURN statement completes execution of a subroutine or function (see "2.397 RETURN Statement").

REWIND

The REWIND statement positions the specified file at its initial point (see "2.398 REWIND Statement").

SELECT CASE

The SELECT CASE statement begins a CASE construct. The CASE construct selects for execution at most one of its constituent blocks (see "2.56 CASE Construct").

SELECT TYPE

The SELECT TYPE statement begins a SELECT TYPE construct. The SELECT TYPE construct selects for execution at most one of its constituent blocks (see "2.413 SELECT TYPE Construct").

STOP

The STOP statement terminates execution of the program (see "2.443 STOP Statement").

Type-guard Statement

The type guard statement is specified in the SELECT TYPE construct and specifies the execution conditions for the block that immediately follows (see "2.413 SELECT TYPE Construct").

SYNC ALL

The SYNC ALL statement causes synchronization among all images (see "2.449 SYNC ALL Statement").

SYNC IMAGES

The SYNC IMAGES statement does synchronization between images (see "2.450 SYNC IMAGES Statement").

SYNC MEMORY

The SYNC MEMORY statement terminates a segment and starts another segment (see "2.451 SYNC MEMORY Statement").

UNLOCK

The UNLOCK statement unlocks lock variables (see "2.477 UNLOCK Statement").

WAIT

The WAIT statement performs a wait operation for the specified unfinished asynchronous data transfer operation (see "2.485 WAIT Statement").

WHERE

The WHERE statement is used to mask the assignment of values in single array assignment statement (see "2.488 WHERE Statement").

WHERE construct

The WHERE construct statement begins a WHERE construct. The WHERE construct controls which elements of an array will be affected by a block of assignment statements (see "2.486 WHERE Construct").

WRITE

The WRITE statement transfers values to an input/output unit from the entities specified in an output list or a namelist group (see "2.489 WRITE Statement").

1.9.2 Nonexecutable Statements

ALLOCATABLE

The ALLOCATABLE statement declares allocatable variables (see "2.15 ALLOCATABLE Statement").

ASYNCHRONOUS

The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute for listed objects (see "2.30 ASYNCHRONOUS Statement").

AUTOMATIC

The AUTOMATIC statement declares specified variable to be on the stack (see "2.40 AUTOMATIC Statement").

BIND

The BIND statement specifies the BIND attribute for listed variables or common blocks (see "2.45 BIND Statement").

BLOCK DATA

The BLOCK DATA statement begins a block data program unit (see "2.51 BLOCK DATA Statement" and "1.11.3 Block Data Program Units").

BYTE type declaration

The BYTE type declaration statement declares entities of type one-byte INTEGER (see "2.469 Type Declaration Statement" and "2.54 BYTE Type Declaration Statement").

CHANGEENTRY

The CHANGEENTRY statement changes rules for processing Fortran procedure names (see "2.60 CHANGEENTRY Statement").

CHARACTER type declaration

The CHARACTER type declaration statement declares entities of type CHARACTER (see "2.469 Type Declaration Statement" and "2.62 CHARACTER Type Declaration Statement").

CLASS

The CLASS type declaration statement declares it as a polymorphic object (see "2.469 Type Declaration Statement" and "2.65 CLASS Type Declaration Statement").

CODIMENSION

The CODIMENSION statement declares coarrays (see "2.73 CODIMENSION Statement" and "1.17 Coarray").

COMMON

The COMMON statement provides a global data facility. It specifies blocks of physical storage: called common blocks that can be accessed by any scoping unit in a program (see "2.75 COMMON Statement").

COMPLEX type declaration

The COMPLEX type declaration statement declares names of type COMPLEX (see "2.469 Type Declaration Statement" and "2.78 COMPLEX Type Declaration Statement").

CONTAINS

The CONTAINS statement separates the body of a main program, module, or subprogram from any internal or module subprograms it contains (see "2.81 CONTAINS Statement").

CONTIGUOUS

The CONTIGUOUS statement declares CONTIGUOUS for array pointer or assumed shape array. (see "2.82 CONTIGUOUS Statement").

DATA

The DATA statement provides initial values for variables (see "2.106 DATA Statement").

DIMENSION

The DIMENSION statement specifies the shape of an array (see "2.113 DIMENSION Statement").

DOUBLE PRECISION type declaration

The DOUBLE PRECISION type declaration statement declares names of type double precision REAL (see "2.469 Type Declaration Statement" and "2.117 DOUBLE PRECISION Type Declaration Statement").

END BLOCK DATA

The END BLOCK DATA statement ends a block data program unit (see "2.130 END BLOCK DATA Statement" and "1.11.3 Block Data Program Units").

END ENUM

The END ENUM statement specifies the end of an enumeration body declaration (see "2.133 END ENUM Statement").

END INTERFACE

The END INTERFACE statement ends an interface block (see "2.138 END INTERFACE Statement" and "1.12.7.2 Procedure Interface Block").

END MAP

The END MAP statement ends a block of union declaration (see "1.5.11.1 Derived Type Definition" and "2.139 END MAP Statement").

END MODULE

The END MODULE statement ends a module (see "2.140 END MODULE Statement" and "1.11.2 Modules").

END STRUCTURE

The END STRUCTURE statement ends a derived type definition that begins a STRUCTURE statement (see "1.5.11.1 Derived Type Definition" and "2.143 END STRUCTURE Statement").

END TYPE

The END TYPE statement ends a derived type definition that begins a TYPE statement (see "1.5.11.1 Derived Type Definition" and "2.145 END TYPE Statement").

END UNION

The END UNION statement ends a union declaration (see "1.5.11.1 Derived Type Definition" and "2.146 END UNION Statement").

ENTRY

The ENTRY statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which the ENTRY statement appears (see "2.148 ENTRY Statement").

ENUM

The ENUM statement specifies the start of an enumeration body declaration (see "2.149 ENUM Statement").

ENUMERATOR

The ENUMERATOR statement declares an enumeration body (see "2.150 ENUMERATOR Statement").

EQUIVALENCE

The EQUIVALENCE statement specifies that two or more objects in a scoping unit share the same storage (see "2.153 EQUIVALENCE Statement").

EXTERNAL

The EXTERNAL statement specifies external procedures, dummy procedures, and block data program unit name (see "2.169 EXTERNAL Statement").

FINAL

The FINAL statement defines the final binding (see "2.172 FINAL Statement").

FORMAT

The FORMAT statement provides explicit information that directs the editing between the internal representation of data and the characters that are input or output (see "2.180 FORMAT Statement" and "1.8.1 Format Specification").

FUNCTION

The FUNCTION statement begins a function subprogram, and specifies characteristics of its function result (see "2.190 FUNCTION Statement" and "1.12.1 Function Subprogram").

IMPLICIT

The IMPLICIT statement specifies, for a scoping unit, a type and type parameters for each name beginning with a letter specified in the statement. Alternately, it can specify that no implicit typing is to apply in the scoping unit (see "2.267 IMPLICIT Statement").

IMPORT

The IMPORT statement specifies that the named entities from the host scoping unit are accessible in the interface body by host association (see "2.268 IMPORT Statement").

INTEGER type declaration

The INTEGER type declaration statement declares names of type INTEGER (see "2.469 Type Declaration Statement" and "2.274 INTEGER Type Declaration Statement").

INTENT

The INTENT statement specifies the intended use of a dummy argument (see "2.275 INTENT Statement").

INTERFACE and ABSTRACT INTERFACE

The INTERFACE statement and ABSTRACT INTERFACE statement begin an interface block. An interface block specifies the forms of reference through which a procedure can be invoked. An interface block can be used to specify a procedure interface, a defined operation, or a defined assignment (see "2.276 INTERFACE Statement" and "1.12.7.2 Procedure Interface Block").

INTRINSIC

The INTRINSIC statement specifies a list of names that represent intrinsic procedures. Doing so permits a name that represents a specific intrinsic function to be used as an actual argument (see "2.277 INTRINSIC Statement").

LOGICAL type declaration

The LOGICAL type declaration statement declares names of type LOGICAL (see "2.469 Type Declaration Statement" and "2.316 LOGICAL Type Declaration Statement").

MAP

The MAP statement begins a block of union declaration (see "1.5.11.1 Derived Type Definition" and "2.324 MAP Statement").

MODULE

The MODULE statement begins a module (see "2.339 MODULE Statement" and "1.11.2 Modules").

NAMELIST

The NAMELIST statement specifies a list of variables which can be referred to by namelist group name in namelist input/output statement (see "2.344 NAMELIST Statement").

OPTIONAL

The OPTIONAL statement specifies that any of the dummy arguments specified need not be associated with an actual argument when the procedure is invoked (see "2.355 OPTIONAL Statement").

PARAMETER

The PARAMETER statement specifies named constants (see "2.358 PARAMETER Statement").

POINTER

The POINTER statement specifies pointers (see "2.361 POINTER Statement" and "1.5.14 Pointer").

POINTER (CRAY Pointer)

The CRAY POINTER statement establishes pair of pointer-based variables and pointer variables (see "2.362 POINTER Statement (CRAY Pointer)").

PRIVATE

The PRIVATE statement in a specification part of a module specifies that the names of entities are accessible only within the current module (see "2.370 PRIVATE Statement"). The PRIVATE statement in a derived type definition specifies that the component names for the type are accessible only within the nodule containing the definition (see "1.5.11.1 Derived Type Definition").

PROCEDURE

The PROCEDURE statement specifies procedure component definition statement, procedure binding statement, procedure declaration statement (see "2.372 PROCEDURE Statement" and "1.12.7 Procedure Interfaces").

PROGRAM

The PROGRAM statement begins the main program (see "2.375 PROGRAM Statement" and "1.11.1 Main Program").

PROTECTED

The PROTECTED statement specifies that the named variable is specified in a MODULE program unit, the part in the useful range by use association where the variable can be modified is limited (see "2.377 PROTECTED Statement").

PUBLIC

The PUBLIC statement specifies that the names of entities are accessible anywhere the module in which the PUBLIC statement appears is used (see "2.378 PUBLIC Statement").

REAL type declaration

The REAL type declaration statement declares names of type REAL (see "2.469 Type Declaration Statement" and "2.391 REAL Type Declaration Statement").

RECORD

The RECORD statement declares names of derived type (see "2.392 RECORD Statement").

SAVE

The SAVE statement specifies that all objects in the statement retain their association, allocation, definition, and value after execution of a RETURN or END statement (see "2.404 SAVE Statement").

SEQUENCE

The SEQUENCE statement can only appear in a derived type definition. It specifies that the order of the component definitions is the storage sequence for objects of that type (see "2.414 SEQUENCE Statement").

Statement Function (obsolescent feature)

A statement function is a function defined by a single statement (see "2.441 Statement Function Statement (obsolescent feature)").

STATIC

The STATIC statement declares specified variable to be in static memory (see "2.442 STATIC Statement").

STRUCTURE

The STRUCTURE statement begins the derived type definition (see "1.5.11.1 Derived Type Definition" and "2.445 STRUCTURE Statement").

SUBROUTINE

The SUBROUTINE statement begins a subroutine subprogram and specifies characteristics of the subroutine (see "2.446 SUBROUTINE Statement" and "1.12.2 Subroutine Subprogram").

TARGET

The TARGET statement specifies a list of object names that have the target attribute and thus can have pointers associated with them (see "2.458 TARGET Statement").

Type Declaration

The type declaration statement declares the type, type parameters, and attributes of data objects (see "2.469 Type Declaration Statement").

TYPE (derived-type definition)

The TYPE statement begins a derived type definition (see "1.5.11.1 Derived Type Definition" and "2.470 TYPE Statement (Derived Type Definition)").

TYPE type declaration

The TYPE type declaration statement declares names of derived type (see "2.471 TYPE Type Declaration Statement").

UNION

The UNION statement begins a union declaration (see "1.5.11.1 Derived Type Definition" and "2.475 UNION Statement").

USE

The USE statement specifies that a specified module is accessible by the current scoping unit (see "2.479 USE Statement").

VALUE

The VALUE statement directs the dummy argument to be associated with a definable temporary area whose initial value is that of the actual argument. Neither the value nor the changes in the definition status for the dummy argument influence the actual argument (see "2.481 VALUE Statement").

VOLATILE

The VOLATILE statement declares specified objects are prevented optimization (see "2.483 VOLATILE Statement").

1.9.3 Statement Order

There are restrictions on where a given statement can appear in a program unit or subprogram. In general,

- USE statements come before specification statements;
- between USE and CONTAINS statements, specification statements appear before executable statements, but FORMAT, DATA, and ENTRY statements can appear among the executable statements; and
- module procedures and internal procedures appear following a CONTAINS statement.

The following table summarizes statement order rules. Vertical lines separate statements that can be interspersed. Horizontal lines separate statements that cannot be interspersed.

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE,
or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td colspan="3">IMPORT statements</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
statements</td><td>IMPLICIT
statements</td></tr>
<tr><td>PARAMETER
and DATA
statements</td><td>Derived-type definitions,
interface blocks,
type declaration statements,
ENUM statements,
procedure declaration statements,
statement function statements,
and specification statements</td></tr>
<tr><td>DATA statements
(obsolescent feature)</td><td>Executable statements</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
</table>

Internal subprograms or module subprograms
END statement

Statements are restricted in what scoping units (see "1.14 Scope") they may appear, as following table:

Kind of scoping unit	Main program	Module	Block data	External sub-program	Module sub-program	Internal sub-program	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
PARAMETER, IMPLICIT, type declaration, and specification statements	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 statement	Yes	No	No	Yes	Yes	Yes	No

1.10 Constructs

Executable constructs control the execution of blocks of statements and nested constructs.

- The IF construct control whether a block will be executed (see "2.263 IF Construct").
- The CASE construct control whether a block will be executed (see "2.56 CASE Construct").
- The DO construct controls how many times a block will be executed (see "2.114 DO Construct").
- The WHERE construct controls which elements of an array will be affected by a block of assignment statements (see "2.486 WHERE Construct").
- The FORALL construct controls multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements (see "2.176 FORALL Construct").
- The ASSOCIATE construct associates an expression or variable with a named element during block execution (see "2.28 ASSOCIATE Construct").
- The SELECT TYPE construct The SELECT TYPE construct enables execution to be selected based on the dynamic type of the expression (see "2.413 SELECT TYPE Construct").
- The BLOCK construct is an executable construct that may include declarations (see "2.50 BLOCK Construct").
- The CRITICAL construct prevents more than one image from executing a block at a same time (see "2.96 CRITICAL Construct").

1.10.1 Construct Names

A constructor name can be specified in the structure construct. If it is specified, it must be specified in the first statement in the structure construct and also in the corresponding last statement. If there is no constructor name specified in a statement in the structure construct, the statement belongs to the innermost structure construct. If a constructor name is specified, the statement belongs to the structure construct having that name. For example, by specifying the constructor names of a CYCLE statement and an EXIT statement, they can belong to a DO construct other than the innermost one, that is, to a nested outside DO construct, making it possible to jump the loop range of that DO construct or end a loop.

1.11 Program Units

Program units are the smallest elements of a Fortran program that may be separately compiled. There are five kinds of program units:

- Main Program
- External Function Subprogram
- External Subroutine Subprogram
- Module Program Unit
- Block Data Program Unit

External Functions and Subroutines are described in "1.12.1 Function Subprogram" and "1.12.2 Subroutine Subprogram".

1.11.1 Main Program

Fortran program shall contain exactly one main program. Execution of a Fortran program begins with the first executable statement in the main program and ends with a STOP statement anywhere in the program or with the END statement of the main program.

The form of a main program is

```
[ PROGRAM program-name ]
    [ use-stmts ]
    [ specification-part ]
    [ execution-part ]
    [ internal-subprogram-part ]
END [ PROGRAM [ program-name ] ]
```

Where:

program-name is the name of the main program.

use-stmts is one or more USE statements.

specification-part is one or more specification statements other than OPTIONAL, INTENT, PUBLIC, PRIVATE, and VALUE statement, derived type definitions, or interface blocks.

execution-part is one or more executable statements, other than RETURN or ENTRY statements.

internal-subprogram-part is CONTAINS statement, followed by one or more internal procedures.

If the *program-name* is specified in the END PROGRAM statement, it shall be identical to the *program-name* specified in the PROGRAM statement.

An automatic object shall not be declared in a main program.

1.11.2 Modules

A module contains specifications and definitions that are to be accessible to other program unit. A module may contain declaration of named variables and named constants, derived type definitions, interface blocks, executable code in module subprograms, and references to other modules. The names in a module can be specified PUBLIC (accessible wherever the module is used) or PRIVATE (accessible only in the scope of the module itself).Variable common block, or procedure pointer have the implicit SAVE attribute in the declaration part of module. Typical uses of modules include

- declaration and initialization of data to be used in more than one program unit.

- specification of explicit interfaces for external subprograms.
- definition of derived types and creation of reusable abstract data types (derived types and the procedures that operate on them).

The form of a module program unit is

```
MODULE module-name
    [ use-stmts ]
    [ specification-part ]
    [ module-subprogram-part ]
END [ MODULE [ module-name ] ]
```

Where:

module-name is the name of the module.

use-stmts is one or more USE statements.

specification-part is one or more specification statements other than OPTIONAL, INTENT, and VALUE statement, derived type definitions, or interface blocks.

module-subprogram-part is CONTAINS statement, followed by one or more module procedures.

If the *module-name* is specified in the END MODULE statement, it shall be identical to the *module-name* specified in MODULE statement.

Example:

```
module example
  implicit none
  integer, dimension(2,2) :: bar1=1, bar2=2
  type phone_number                    !derived type definition
    integer :: area_code,number
  end type phone_number
  interface                            !explicit interfaces
    function test(sample,result)
      implicit none
      real :: test
      integer, intent(in) :: sample,result
    end function test
    function count(total)
      implicit none
      integer :: count
      real,intent(in) :: total
    end function count
  end interface
  interface swap                       !generic interface
    module procedure swap_reals,swap_integers
  end interface
  contains
    subroutine swap_reals(r1,r2)       !module procedure
      real,intent(inout) :: r1,r2
      real :: t
      t = r1 ; r1 = r2 ; r2 = t
    end subroutine swap_reals
    subroutine swap_integers(i1,i2)    !module procedure
      integer,intent(inout) :: i1,i2
      integer :: t
      t = i1 ; i1 = i2 ; i2 = t
    end subroutine swap_integers
end module example
```

1.11.2.1 Module Procedures

Module procedures have the same rules and organization as external procedures (see "1.12.1 Function Subprogram" and "1.12.2 Subroutine Subprogram"). Only program units that use the host module have access to the module's module procedures. Procedures may be made local to the module by specifying the PRIVATE attribute in a PRIVATE statement or in a type declaration statement within the module.

1.11.2.2 Using Modules

Information contained in a module may be made available within another program unit via the USE statement, it called use association.

For example,

```
use mod_sample
```

would give the current scoping unit access to the names in module `mod_sample`. If a name in `mod_sample` conflicts with a name in the current scoping unit, an error occurs only if that name is referenced. To avoid such conflicts, the USE statement has an aliasing facility:

```
use mod_sample, a => b
```

Here the module entity `b` would be known as `a` in the current scoping unit.

Another way of avoiding name conflicts, if the module entity name is not needed in the current scoping unit, is with the ONLY form of the USE statement:

```
use mod_sample, only : c, d
```

Here, only the names `c` and `d` are accessible to the current scoping unit.

1.11.3 Block Data Program Units

A block data program unit provides initial values for data in one or more named common blocks. Only specification statements may appear in a block data program unit.

The form of a block data program unit is

```
BLOCK DATA [ block-data-name ]
    [ use-stmts ]
    [ specification-part ]
END [ BLOCK DATA [ block-data-name ] ]
```

Where:

block-data-name is the name of the block data program unit. *block-data-name* in the BLOCK DATA statement may be not specified. However there shall not be more than one unnamed block data program unit in a program.

use-stmts is one or more USE statements.

specification-part is one or more specification statements other than ALLOCATABLE, INTENT, PUBLIC, PRIVATE, OPTIONAL, EXTERNAL, and VALUE statement, or derived type definitions.

If the *block-data-name* is specified in the END BLOCK DATA statement, it shall be identical to the *block-data-name* specified in BLOCK DATA statement.

1.12 Procedures

A procedure encapsulates an arbitrary sequence of computations that may be invoked directly during program execution. Procedures are either functions or subroutines. A function is a procedure that is invoked in an expression; its invocation causes a value to be computed which is then used in evaluating the expression. A subroutine is a procedure that is invoked in a CALL statement or by a defined assignment statement.

A procedure classified follows by means of definition:

Intrinsic procedure

An intrinsic procedure is a procedure that is provided as an inherent of the processor. Each is documented in detail in the "Chapter 2 Alphabetical Reference". A table is provided in " Appendix A Intrinsic Procedures".

External procedure

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

Module procedure

A module procedure is a procedure that is defined by a module subprogram. A module procedure may be invoked by another module subprogram in the module or by any scoping unit that access the module procedure by use association.

Internal procedure

An internal procedure is a procedure that is defined by an internal subprogram. The containing main program or subprogram is called the host of the internal procedure. The internal procedure is accessible within the scoping units of the host and all its other internal procedures but is not accessible elsewhere.

Dummy procedure

A dummy procedure is a dummy argument that is specified as a procedure or appears in a procedure reference. The actual argument that is associated with dummy procedure shall be the specific name of an external, module, dummy, or intrinsic procedure. Executing the dummy procedure reference invoked the procedure supplied as the actual argument corresponding to the dummy procedure.

Procedure pointer that is not a dummy argument

A procedure pointer can pointer associate with a dummy procedure that is not an external procedure, a module procedure, an intrinsic procedure, or a procedure pointer.

Statement function (obsolescent feature)

A function defined by a single statement is called a statement function.

1.12.1 Function Subprogram

A function subprogram defines an external function subprogram, module function subprogram, or internal function subprogram.

The syntax for a function subprogram definition is

```
[ prefix-spec ] ... FUNCTION function-name ( [ dummy-arg-name-list ] ) [ suffix ]
    [ use-stmts ]
    [ specification-part ]
    [ execution-part ]
    [ internal-subprogram-part ]
END [ FUNCTION [ function-name ] ]
```

Where:

prefix-spec is

```
type-spec       or
RECURSIVE       or
PURE            or
ELEMENTAL
```

type-spec is a declaration type specifier (see "2.469 Type Declaration Statement").

function-name is the name of the function.

dummy-arg-name-list is a comma-separated list of the dummy argument name.

suffix is

```
proc-language-binding-spec [ RESULT ( result-name ) ]    or
RESULT ( result-name ) [ proc-bind-spec ]
```

result-name is the name of the result variable. If RESULT is not specified, the result variable is *function-name*.

proc-language-binding-spec is

```
BIND ( C [, NAME = scalar-char-initialization-expr ] )
```

scalar-char-initialization-expr is scalar CHARACTER constant expression.

use-stmts is one or more USE statements.

specification-part is one or more specification statements other than PUBLIC and PRIVATE statement, derived type definitions, or interface blocks.

execution-part is one or more executable statements.

internal-subprogram-part is CONTAINS statement, followed by one or more internal procedures.

FUNCTION shall be present in the END FUNCTION statement of an internal or module function.

If the *function-name* is specified in the END FUNCTION statement, it shall be identical to the *function-name* specified in FUNCTION statement.

An internal function subprogram shall not contain an ENTRY statement and an *internal-subprogram-part*.

For example, in the following program,

```
program main
  implicit none
  interface ! an explicit interface is provided
    function square(x)
      implicit none
      real, intent(in) :: x
      real :: square
    end function square
  end interface
  real :: a, b=3.6, c=3.8
  a = 3.7 + b + square(c) + sin(4.7)
  print *, a
  stop
end program main

function square(x)
  implicit none
  real, intent(in) :: x
  real :: square
  square = x*x
  return
end function square
```

square(c) and sin(4.7) are function references.

1.12.2 Subroutine Subprogram

A subroutine subprogram defines an external subroutine subprogram, module subroutine subprogram, or internal subroutine subprogram.

The syntax for a subroutine definition is

```
[ prefix-spec ] ... SUBROUTINE subroutine-name [ ( [ dummy-arg-list ] ) proc-language-binding-spec ]
    [ use-stmts ]
    [ specification-part ]
    [ execution-part ]
    [ internal-subprogram-part ]
END [ SUBROUTINE [ subroutine-name ] ]
```

Where:

prefix-spec is

```
RECURSIVE           or
PURE                or
ELEMENTAL
```

subroutine-name is the name of the subroutine.

dummy-arg-list is a comma-separated list of

```
dummy-arg-name    or
*
```

dummy-arg-name is the name of the dummy argument.

proc-language-binding-spec is

```
BIND ( C [, NAME = scalar-char-initialization-expr ] )
```

scalar-char-initialization-expr is scalar CHARACTER constant expression.

use-stmts is one or more USE statements.

specification-part is one or more specification statements other than PUBLIC and PRIVATE statement, derived type definitions, or interface blocks.

execution-part is one or more executable statements.

internal-subprogram-part is CONTAINS statement, followed by one or more internal procedures.

SUBROUTINE shall be present in the END SUBROUTINE statement of an internal or module subroutine.

If the *subroutine-name* is specified in the END SUBROUTINE statement, it shall be identical to the *subroutine-name* specified in SUBROUTINE statement.

An internal subroutine subprogram shall not contain an ENTRY statement and an *internal-subprogram-part*.

For example,

```
program main
  implicit none
  interface ! an explicit interface is provided
    subroutine multiply(x, y)
      implicit none
      real, intent(in out) :: x
      real, intent(in) :: y
    end subroutine multiply
  end interface
  real :: a, b
  a = 4.0
  b = 12.0
  call multiply(a, b)
  print *,a
end program main

subroutine multiply(x, y)
  implicit none
  real, intent(in out) :: x
  real, intent(in) :: y
  x = x*y
end subroutine multiply
```

This program calls the subroutine `multiply` invoked in a CALL statement.

1.12.3 Recursive Procedures

A Fortran procedure can reference itself or a procedure defined by an ENTRY statement in the same subprogram, either directly or indirectly, only if the RECURSIVE keyword is specified in the FUNCTION or SUBROUTINE statement (see "2.190 FUNCTION

Statement" or "2.446 SUBROUTINE Statement"). A function that calls itself or a function defined by an ENTRY statement in the same subprogram directly shall use the RESULT option in the FUNCTION or ENTRY statement.

1.12.4 Pure Procedures

A pure subprogram is a subprogram that has the PURE or ELEMENTAL keyword in the FUNCTION or SUBROUTINE statement. A pure subprogram shall not contain any operation that could conceivably result in an assignment or pointer assignment to a common variable, a variable associated by use or host association, a coindexed object, or an INTENT(IN) dummy argument. It means that a pure procedure is free from side effects. A pure procedure has following constraints:

- For a pure function, all dummy arguments shall have INTENT(IN) except procedure arguments and arguments with the POINTER attribute.
- For a pure subroutine, all dummy arguments shall be specified intents except procedure arguments and arguments with the POINTER attribute.
- All dummy arguments of a pure procedure that are procedure arguments shall be pure.
- A local variable in a pure procedure shall not have the SAVE attribute and the explicitly initialization.
- All internal subprograms in a pure subprogram shall be pure.
- In a pure subprogram any variable which is in common or accessed by host or use association, is a dummy argument to a pure function, is a dummy argument with INTENT(IN) to a pure subroutine, or an object that is storage associated with any such variable, shall not be used in the contexts where variables to become defined, pointers to become associated or assumed, and allocation status to be changed.
- Any procedure referenced in a pure subprogram shall be pure.
- Any procedure referenced in a pure subprogram, including one referenced via a defined operation, defined assignment or finalization, shall be pure.
- A pure subprogram shall not contain any input/output statements except internal-file input/output.
- A pure subprogram shall not contain a STOP statement or ERROR STOP statement.
- A pure subprogram shall not contain an image control statement (see "1.18 Image Control Statement").

1.12.5 Elemental Procedures

An elemental procedure is one that processes separately each array element specified in the arguments.

1.12.5.1 Elemental Procedure Declaration

An elemental procedure is a procedure defined by an elemental intrinsic procedure or an elemental subroutine program.

If the prefix specifier ELEMENTAL is specified in prefix-spec of the FUNCTION statement and SUBROUTINE statement, that procedure becomes an elemental procedure.

Since it is an inclusion that the prefix specifier ELEMENTAL is pure, the prefix specifier PURE need not be written. The following restrictions apply to elemental procedures:

- All dummy arguments of an elemental procedure must be scalar dummy data objects, cannot have the pointer attribute or the ALLOCATABLE attribute, and must not be coarray.
- The result variables of elemental functions must be scalar, and cannot have the pointer attribute or the ALLOCATABLE attribute.
- A dummy argument, or a subobject thereof, shall not appear in a specification expression except as the argument to one of the intrinsic functions BIT_SIZE, KIND, LEN, or the numeric inquiry functions.
- A dummy argument shall not be a dummy procedure.
- A dummy argument shall not be *.

An elemental subprogram is a pure subprogram and all of the constraints for pure subprograms also apply (see "1.12.4 Pure Procedures").

1.12.5.2 Actual Arguments and Results of Elemental Functions

An elemental function is scalar if it is referenced using a generic name or a specific name and the result format does not have an actual argument specification, or if all actual arguments are scalar. If the array appears in an actual argument, it is that format. If an elemental function has two or more arguments, all actual arguments must be format compatible. For an array, the values of each element in the results are the values obtained by applying scalar functions individually (regardless of the order) to the elements in each of the corresponding arguments.

1.12.5.3 Actual Arguments of Elemental Subroutines

If an elemental subroutine is referenced, one of the following must apply:

- All actual arguments are scalar.
- Actual arguments associated with dummy arguments that have the INTENT(OUT) or INTENT(INOUT) attribute are all arrays with the same format and other arguments are format compatible with them.

When actual arguments associated with dummy arguments having the INTENT(OUT) attribute or the INTENT(INOUT) attribute are arrays, the values of each element are obtained by applying scalar value functions individually (regardless of the order) to the elements in each of the corresponding arguments.

1.12.6 Procedure Reference

The syntax for a function reference is

```
procedure-designator ( [ actual-arg-spec-list ] )
```

procedure-designator is designated for the name of the function.

Functions can also be referenced as user-defined operations (see "1.12.7.3.2 Defined Operations"). A function is invoked during expression evaluation. When it is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is available for use in the expression that caused the function to be invoked.

The syntax for a subroutine reference is

```
CALL procedure-designator [ ( [ actual-arg-spec-list ] ) ]
```

procedure-designator is designated for the name of the subroutine.

procedure-designator is

```
procedure-name                or
proc-component-ref            or
data-ref % binding-name
```

procedure-name is a name of a procedure.

proc-component-ref is referenced for procedure component. A procedure component shall be a name of procedure pointer component.

A subroutine is used for defined assignment (see "1.12.7.3.3 Defined Assignment").

data-ref is a data object reference.

binding-name is a name of a binding.

If a data reference is an array, the referencing type bound procedure must have the PASS attribute.

actual-arg-spec-list is a comma-separated list of dummy argument name actual argument specifier.

actual-arg-spec is

```
[ keyword = ] actual-arg
```

keyword is the dummy argument name.

actual-arg is the actual argument as following:

```
expr                    or
variable                or
procedure-name          or
proc-component-ref      or
*label                  or
%VAL( expr )
```

expr is an expression.

variable is a variable.

procedure-name is a name of the procedure. A non-intrinsic elemental procedure shall not be used as an actual argument. A *procedure-name* of *actual-arg* shall not be the name of an internal procedure or of a statement function and shall not be the generic name of a procedure unless it is also a specific name.

proc-component-ref is a procedure component reference.

**label* is an alternate return specifier. *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the CALL statement. **label* shall not be specified as an actual argument of a function reference.

%VAL specifier for actual argument passes argument by value.

1.12.6.1 Procedure Arguments

Arguments provide a means of passing information between a calling procedure and a procedure it calls. The calling procedure provides a list of actual arguments. The called procedure accepts a list of dummy arguments. The type, type parameters, and shape of the actual argument shall be consistent with the characteristics of the dummy arguments.

The actual argument list identifies the correspondence between the actual arguments supplied and the dummy arguments of the procedure. If the absence of an argument keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; that is, the first actual argument is associated with the first dummy argument, the second actual argument is associated with the second dummy argument, etc. If an argument keyword is present, the actual argument is associated with 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). Exactly one actual argument shall be associated with each nonoptional dummy argument. At most one actual argument may be associated with each optional dummy argument. Each actual argument shall be associated with a dummy argument.

1.12.6.1.1 Argument Intent

The INTENT attribute (see "2.275 INTENT Statement") intended use of dummy argument.

The INTENT attribute is specified for dummy arguments using the INTENT statement or INTENT attribute specifier in a type declaration statement.

1.12.6.1.2 Argument Keywords

If an argument keyword *keyword*= is present, the actual argument is associated with the dummy argument whose name is the same as the *keyword*, regardless of position in the actual argument list. If the absence of an argument keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list.

An argument keyword may appear if the interface of the procedure is explicit in the scoping unit (see "1.12.7 Procedure Interfaces"). An argument keyword shall appear if the actual argument may not be associated with the dummy argument occupying the corresponding position in the dummy argument list, or an argument keyword has been appear preceding actual argument specifier in the argument list.

Example:

```
interface
  subroutine zee(a,b,c)
    integer :: a,b,c
  end subroutine
end interface
call zee(c=1, b=2, a=3)
```

In the example, the actual arguments are provided in reverse order.

1.12.6.1.3 Optional Arguments

An actual argument need not be provided for a corresponding dummy argument with the OPTIONAL attribute. A dummy argument is not present if the dummy argument has the following conditions:

- The dummy argument does not correspond to an actual argument. or,
- The dummy argument corresponds to an actual argument and the dummy argument is not pointer or allocatable. and,
  - The actual argument is an allocatable entity and the allocatable entity is not allocated. or,
  - The actual argument is a pointer entity and the pointer entity is disassociated.

To make an argument optional, specify the OPTIONAL attribute for the dummy argument, either in a type declaration statement or with the OPTIONAL statement.

A procedure that has an optional dummy argument shall have an explicit interface (see "1.12.7 Procedure Interfaces").

The PRESENT intrinsic function takes as an argument the name of an optional argument and returns true if the argument is present and false otherwise. A dummy argument or procedure that is not present shall not be referenced except as an argument to the PRESENT function or as an optional argument in a procedure reference.

An optional argument at the end of a dummy argument list can simply be omitted from the corresponding actual argument list. An argument keyword (see "1.12.6.1.2 Argument Keywords") shall appear to omit other optional arguments, unless all of the remaining arguments in the reference are omitted.

For example,

```
subroutine zee(a, b, c)
  implicit none
  real, intent(in), optional :: a, c
  real, intent(in) :: b
end subroutine zee
```

In the above subroutine, a and c are optional arguments. In the following calls, various combinations of optional arguments are omitted:

```
call zee(b=3.0)          ! a and c omitted, keyword necessary
call zee(2.0, 3.0)       ! c omitted, keywords may be omitted
call zee(b=3.0, c=8.5)   ! a omitted, keywords necessary
```

1.12.6.1.4 Dummy Data Objects

The type parameter values of the actual argument must agree with the corresponding ones of the dummy argument that are not assumed or deferred, except for the case of the character length parameter of an actual argument of type character associated with a dummy argument that is not assumed shape.

If a dummy argument is a dummy data object, the associated actual argument shall be an expression of the same type or a data object of the same type. The kind type parameter value of the actual argument shall agree with that of the dummy argument.

If the dummy argument is a pointer, the actual argument shall be a pointer and the types, type parameters, and ranks shall agree. At the invocation of the procedure, the dummy argument pointer receives the pointer association status of the actual argument. If the actual argument is currently associated, the dummy argument becomes associated with the same target. The association status may change during the execution of the procedure. When execution of the procedure completes, the pointer association status of the dummy argument become undefined if it is associated with a target that becomes undefined, the pointer association status of the actual argument becomes that of the dummy argument.

If the dummy argument is a nonoptional, nonpointer, and the corresponding actual argument is a pointer, the actual argument shall be currently associated with a target and the dummy argument becomes argument associated with that target.

If the dummy argument is a nonoptional, nonallocatable, and the corresponding actual argument is an allocatable, the actual argument shall be allocated.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure.

If the dummy argument has the TARGET attribute and is either a scalar or an assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, any pointers associated with the actual argument become

associated with the corresponding dummy argument on invocation of the procedure, and when execution of the procedure completes, any pointers associated with the dummy argument remain associated with the actual argument.

If the dummy argument has the TARGET attribute and is an explicit-shape array or is an assumed-size array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure, and when execution of the procedure completes, any pointers associated with the dummy argument remain associated with the actual argument.

If the dummy argument has the TARGET attribute and the corresponding actual argument does not have the TARGET attribute or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

If the actual argument is scalar, the corresponding dummy argument shall be scalar unless the actual argument is an element of an array that is not an assumed-shape or pointer array, a substring of such an element, or a character scalar. If the procedure is nonelemental and is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments shall agree.

If a dummy argument is an assumed-shape array, the actual argument shall not be an assumed-size array or a scalar.

If the actual argument is a coindexed scalar, the corresponding dummy argument must be a scalar.

A scalar dummy argument of a nonelemental procedure may be associated only with a scalar actual argument.

Dummy data object characteristics can be type, type parameter, format, INTENT attribute, OPTIONAL attribute, ALLOCATABLE attribute, VALUE attribute, ASYNCHRONOUS attribute, VOLATILE attribute, polymorphic, POINTER attribute, CONTIGUOUS attribute, and TARGET attribute. If object type parameters or array bounds are initialization expressions, methods that depend on elements of those expressions are also characteristics. If the format or size is inherited, or if the type parameter is not specified, these are also characteristics.

If a dummy argument is allocatable or a polymorphic pointer, the associated actual argument must be polymorphic. In addition, one of the following must apply:

- The actual argument and the dummy argument must be unlimited and polymorphic.
- The declared type of the actual argument must be the same as the dynamic type of the dummy argument.

If a dummy argument is allocatable or a pointer, the associated actual argument type parameter that is the same as that of the dummy argument must be a deferred type.

If a dummy argument is a pointer, the actual argument must be a pointer and deferred type parameters and ranks must be the same.

If a dummy argument is a pointer array with CONTIGUOUS attribute, the actual argument must be a pointer with CONTIGUOUS attribute.

If a dummy argument is allocatable, the actual argument must be allocatable and deferred type parameters and ranks must be the same.

If the dummy argument is a coarray, the corresponding actual argument must be a coarray. The corank of the actual argument must be the same as that of the allocatable coarray dummy argument.

If an actual argument is an array section or an assumed-shape array, and if the corresponding dummy argument has either the VOLATILE attribute or the ASYNCHRONOUS attribute, that dummy argument must be an assumed-shape array.

A dummy argument must be an assumed-shape array that does not have the CONTIGUOUS attribute under the following conditions:

- The actual argument is not simply contiguous, nonpointer, and has either the VOLATILE attribute or the ASYNCHRONOUS attribute. and,
- The corresponding dummy argument has either the VOLATILE attribute or the ASYNCHRONOUS attribute.

If an actual argument is a pointer array, and if the corresponding dummy argument has either the VOLATILE argument or the ASYNCHRONOUS argument, that dummy argument must be an assumed-shape array or a pointer array.

A dummy argument must be an assumed-shape or an array pointer that does not have the CONTIGUOUS attribute under the following conditions:

- The actual argument has either the VOLATILE attribute or the ASYNCHRONOUS attribute, and an array pointer that does not have the CONTIGUOUS attribute. and,
- The corresponding dummy argument has either the VOLATILE attribute or the ASYNCHRONOUS attribute.

An actual argument that is a coindexed object must not have a pointer ultimate component or allocatable ultimate component. If the actual argument is a coindexed object, the dummy argument must have the INTENT (IN) attribute.

If dummy argument is an array coarray that has the CONTIGUOUS attribute or that is not of assumed shape, its corresponding actual argument must be simply contiguous.

1.12.6.1.5 Dummy Procedures

A dummy procedure is a dummy argument that is specified as a procedure with EXTERNAL attribute or interface body, or appears in a procedure reference.

If a dummy argument is a dummy procedure, the associated actual argument shall be the specific name of an external, module, dummy, or intrinsic procedure.

If a dummy argument is a procedure pointer, the associated actual argument must be a procedure pointer, a function reference that returns a procedure pointer, or an intrinsic function NULL reference.

1.12.6.1.6 Alternate Returns (obsolescent feature)

If a dummy argument is an asterisk, the associated actual argument shall be an alternate return specifier **label*. *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the CALL statement. An alternate return specifier specifies the statement that transfers control to after the subroutine is executed. **label* shall not be specified as an actual argument of a function reference.

1.12.7 Procedure Interfaces

The interface of a procedure determines the forms of reference through which it may be invoked. The interface consists of the characteristics of the procedure, the name of the procedure, the name and characteristics of each dummy argument, and the procedure's generic interfaces, if any. The characteristics of a procedure are the classification of the procedure as a function or subroutine, whether or not it is pure, whether or not it is elemental, the characteristics of its dummy arguments, and the characteristics of its result value if it is a function. The characteristics of a dummy argument is either a dummy argument, a dummy procedure, or an asterisk. The characteristics of a dummy data object are its type, its type parameters, its shape, its intent, whether it is optional, whether it is allocatable, whether it has the VALUE attribute, whether it has the CONTIGUOUS attribute, whether it has the VOLATILE attribute, whether it is polymorphic, and whether it is a pointer or a target. The characteristics of a dummy procedure are the explicitness of its interface, its characteristics as a procedure if the interface is explicit, and whether it is optional. The characteristics of a function result are its type, type parameters, rank, and whether it is allocatable or a pointer.

If a procedure is accessible in a scoping unit, its interface is either explicit or implicit in that scoping unit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit in such a scoping unit. The interface of a recursive subroutine or a recursive function with a separate result name is explicit within the subprogram that defines it. The interface of an external procedure or dummy procedure is explicit in a scoping unit other than its own if an interface block for the procedure is supplied or accessible, and implicit otherwise.

1.12.7.1 Explicit Interfaces

A procedure shall have an explicit interface if

a reference to the procedure appears

- with an argument keyword, or
- as a reference by its generic name, or
- as a defined assignment, or
- in an expression as a defined operator, or
- in a context that requires it to be pure,

the procedure has

- an optional dummy argument, or
- a procedure with the BIND attribute, or
- a dummy argument that is an allocatable, an assumed-shape array, a pointer, or a target, or

- an array-valued result, or
- a result that is a pointer or an allocatable, or
- a result whose CHARACTER length parameter or type parameter is neither assumed nor initialization expression, or
- a dummy argument that has the VOLATILE, VALUE or ASYNCHRONOUS attribute, or the procedure is elemental, or
- a dummy argument of a derived type with parameters, or
- a polymorphic dummy argument, or
- a coarray dummy argument.

1.12.7.2 Procedure Interface Block

The syntax for an interface block is

```
INTERFACE [ generic-spec ]
    [ interface-body ] ...
    [ [ MODULE ] PROCEDURE [::] procedure-name-list ] ...
END INTERFACE [ generic-spec ]
```

or

```
ABSTRACT INTERFACE
    [ interface-body ] ...
END INTERFACE
```

Where:

generic-spec is

```
generic-name                        or
OPERATOR ( defined-operator )       or
ASSIGNMENT (=)                      or
dtio-generic-spec
```

generic-name is the name of the generic procedure.

defined-operator is

```
intrinsic-operator    or
. operator-name .
```

intrinsic-operator is an intrinsic operator.

operator-name is a user-defined name for the operation, consisting of one to 240 letters.

ASSIGNMENT(=) is a user-defined assignment.

dtio-generic-spec is a derived type input/output procedure, is

```
READ ( FORMATTED )          or
READ ( UNFORMATTED )        or
WRITE ( FORMATTED )         or
WRITE ( UNFORMATTED )
```

procedure-name-list is comma-separated list of procedure name.

A *procedure-name* must have an explicit interface and must specify a procedure pointer, external procedure, dummy procedure or module procedure.

If MODULE appears in a procedure statement, each *procedure-name* must be a module procedure.

If the *generic-spec* is specified in the END INTERFACE statement, it shall be identical to the *generic-spec* specified in INTERFACE statement. But, *generic-spec* is not specified in the ABSTRACT INTERFACE statement.

interface-body is

```
[ prefix-spec ] ... FUNCTION function-name ( [ dummy-arg-name-list ] ) [ suffix ]
    [ use-stmts ]
    [ import-stmts ]
    [ specification-part ]
END [ FUNCTION [ function-name ] ]

or

[ prefix-spec ] ... SUBROUTINE subroutine-name [ ( [ dummy-arg-list ] ) proc-language-binding-spec ]
    [ use-stmts ]
    [ import-stmts ]
    [ specification-part ]
END [ SUBROUTINE [ subroutine-name ] ]
```

prefix-spec is

```
type-spec    or
RECURSIVE    or
PURE         or
ELEMENTAL
```

type-spec is the type declaration specifier (see "2.469 Type Declaration Statement"). *type-spec* shall not be specified in the SUBROUTINE statement.

function-name is the name of the function.

suffix is

```
proc-language-binding-spec [ RESULT ( result-name ) ]    or
RESULT ( result-name ) [ proc-language-binding-spec ]
```

result-name is the name of the result variable. If RESULT is not specified, the result variable is *function-name*.

proc-language-binding-spec is the following procedure language binding specifier.

```
BIND ( C [, NAME = scalar-char-initialization-expr ]
```

scalar-char-initialization-expr is a scalar CHARACTER constant expression.

subroutine-name is the name of the subroutine.

dummy-arg-name-list is a comma-separated list of dummy argument name.

dummy-arg-list is a comma-separated list of

```
dummy-arg-name    or
*
```

dummy-arg-name is a named of dummy argument.

use-stmts is one or more USE statements.

import-stmts is one or more IMPORT statements.

specification-part is one or more specification statements other than PUBLIC and PRIVATE statement, derived type definitions, or interface blocks. The ENTRY, DATA, and FORMAT statement shall not appear in *specification-part*.

If the *function-name* is specified in the END FUNCTION statement, it shall be identical to the *function-name* specified in FUNCTION statement.

If the *subroutine-name* is specified in the END SUBROUTINE statement, it shall be identical to the *subroutine-name* specified in SUBROUTINE statement.

An interface body in an interface block specifies an explicit specific interface for an existing external procedure or a dummy procedure. If the name in a FUNCTION or SUBROUTINE statement in an interface block is the same as the name of a dummy argument in the subprogram containing the interface block, the interface block declares that dummy argument to be a dummy procedure with the indicated interface; otherwise, the interface block declares the name to be the name of an external procedure with the indicated procedure interface.

An interface body specifies all of the procedure's characteristics and these shall be consistent with those specified in the procedure definition, except that the interface may specify a procedure that is not pure if the procedure is defined to be pure. An interface block shall not contain an ENTRY statement, but an entry interface may be specified by using the entry name as the procedure name in the interface body. A procedure shall not have more than one explicit specific interface in a given scoping unit.

If the INTERFACE statement is ABSTRACT INTERFACE, the function name in the FUNCTION statement and the subroutine name in the SUBROUTINE statement cannot be the same as the keyword specifying the intrinsic type.

An abstract interface block uses ABSTRACT INTERFACE. The interface body within the abstract interface block declares the abstract interface.

An interface block with a generic specifier is a generic interface block.

An interface block specifying neither ABSTRACT nor a generic specifier is a specific interface block.

Example:

```
interface
  subroutine x(a, b, c)
    implicit none
    real, intent(in), dimension (2,8) :: a
    real, intent(out), dimension (2,8) :: b, c
  end subroutine x
  function y(a, b)
    implicit none
    integer :: y
    logical, intent(in) :: a, b
  end function y
end interface
```

In this example, explicit interfaces are provided for the procedures `x` and `y`.

1.12.7.3 Generic Interfaces

An INTERFACE statement may be contained a *generic-spec*. An interface block with a generic specification specifies a generic interface for each of the procedures in the interface block. A PROCEDURE statement is allowed only if the INTERFACE statement has a *generic-spec*. All procedure names in a PROCEDURE statement shall have an explicit interface. And it shall be a procedure pointer, external procedure, dummy procedure, or module procedure.

A procedure reference to the generic interface shall be consistent with exactly one specific procedure; the reference is to the specific procedure.

1.12.7.3.1 Generic Names

A generic name in an INTERFACE statement specifies a single name to reference all if the procedure names in the interface block. A generic name may be the same as any one of the procedure names in the interface block, or the same as any accessible generic name.

Within a scoping unit, two procedures that have the same generic name shall both be subroutines or both be functions, and one of the following must apply:

- Both have differing numbers of dummy arguments.
- One or more dummy arguments have a different type, different kind type parameter, or different rank from those of the other.
- Both have a passed-object dummy argument that can be distinguished.

If a generic name is the same as the name of a generic intrinsic procedure, the generic intrinsic procedure is not accessible if the procedures in the interface and the intrinsic procedure are not all functions or subroutines. If a generic invocation applies to both a specific procedure from an interface and an accessible generic interface procedure, it is the specific procedure from the interface that is referenced.

Example:

```
interface swap ! generic swap routine
  subroutine real_swap(x,y)
    implicit none
    real, intent(inout) :: x,y
  end subroutine real_swap
```

```
  subroutine int_swap(x,y)
    implicit none
    integer, intent(inout) :: x,y
  end subroutine int_swap
end interface
```

Here the generic procedure `swap` can be used with both the REAL and INTEGER types.

1.12.7.3.2 Defined Operations

Operators can be extended and new operators created for user-defined and intrinsic data types. This is done using interface blocks with INTERFACE OPERATOR. All of the procedures specified in that interface block shall be functions. In the case of functions of two arguments, infix binary operator notation is implied. In the case of functions of one argument, prefix operator notation is implied. The dummy arguments shall be nonoptional dummy data objects and shall be specified with INTENT(IN). A defined operation is treated as a reference to the function. For a unary defined operation, the operand corresponds to the function's dummy argument; for a binary operation, the left-hand operand corresponds to the first argument of the function and the right-hand operand corresponds to the second dummy argument.

A defined operation has the form

```
defined-unary-operator operand              or
operand defined-binary-operator operand
```

defined-unary-operator is a unary operator.

defined-binary-operator is a binary operator.

operand is an operand.

defined-unary-operator and *defined-binary-operator* is one of the intrinsic operators or a user-defined operator of the form

```
.operator-name.
```

where *operator-name* consists of one to 240 letters.

One of the following must apply:

- A function with an OPERATOR(.*operator-name*.) generic specifier is specified.
- There is a generic binding with the OPERATOR(.*operator-name*.) generic specifier in the operand declared type, and there is a binding corresponding to the function within that dynamic type.

The dummy argument type of the function must be compatible with the actual argument dynamic type.

The precedence of a defined operator is the same as that of the corresponding intrinsic operator if an intrinsic operator is being extended. If a user-defined operator is used, a unary defined operation has higher precedence than any other operation, and a binary defined operation has a lower precedence than any other operation.

If the generic specification extends an intrinsic operator, each specific procedure in the interface block shall not have the same specification with an intrinsic operation for types, kind type parameters, and rank. Within a scoping unit, if two procedures have the same generic operator and the same number of arguments, one shall have a dummy argument that corresponds by position in the argument list to a dummy argument of the other that has a different type, different kind type parameter, or different rank.

For example,

```
use type_define_mod
type(set) :: a,b,c
interface operator (.intersection.)
  function set_intersection (a, b)
    use type_define_mod
    implicit none
    type (set), intent(in) :: a, b
    type (set) :: set_intersection
  end function set_intersection
end interface operator (.intersection.)
a = b.intersection.c
```

`b.intersection.c` is treated as a function reference `set_intersection(b,c)`.

In other example,

```
use type_define_mod
type(set) :: a,b,c
interface operator (*)
  function set_intersection (a, b)
    use type_define_mod
    implicit none
    type (set), intent (in) :: a, b
    type (set) :: set_intersection
  end function set_intersection
end interface
a = b*c
```

`b*c` is treated as a function reference `set_intersection(b,c)`.

1.12.7.3.3 Defined Assignment

The assignment may be extended using an interface block with INTERFACE ASSIGNMENT. All of the procedures specified in that interface block shall be subroutines. Each of these subroutines shall have exactly two dummy arguments. Each argument shall be nonoptional. The first argument shall have INTENT(OUT) or INTENT (INOUT) and the second argument shall have INTENT(IN). A defined assignment is treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parenthesis as the second argument.

If the generic specification specifies the defined assignment, each specific procedure in the interface block shall not have the same specification with an intrinsic assignment for types, kind type parameters, and rank. Within a scoping unit, if two procedures both define assignment, one shall have a dummy argument that corresponds by position in the argument list to a dummy argument of the other that has a different type, different kind type parameter, or different rank.

A subroutine which defines the defined assignment must be the following either.

- A generic interface provides the subroutine with a generic specifier of ASSIGNMENT(=), or
- There is a generic binding in the declared type with a generic specifier of ASSIGNMENT(=) and there is a corresponding binding to the subroutine in the dynamic type, respectively.

The types of dummy argument in subroutine must be compatible with the dynamic types of the assignment, respectively.

Example:

```
interface assignment (=) ! use = for integer to
                         ! logical array
  subroutine integer_to_logical_array (b, n)
    implicit none
    logical, intent (out) :: b(:)
    integer, intent (in) :: n
  end subroutine integer_to_logical_array
end interface
```

Here the assignment operator is extended to convert INTEGER data to a LOGICAL array.

1.12.7.4 Solving Type Bound Procedure References

If the procedure designator binding name is the name of a specific binding procedure, the referenced procedure is the procedure bound to the same name as the specific binding selected in the data reference dynamic type.

If the procedure designator binding name is the name of a generic binding procedure, a specific binding is selected as follows from the generic bindings with that name in the data reference declared type:

1. If the reference matches one of the specific bindings in the generic bindings, that specific binding is selected.
2. Otherwise, the specific binding that matches the elemental processing reference of one of the specific bindings in the generic bindings is selected for the reference.

1.12.8 Service Routines

Service routines are functions and subroutines that are offered as external procedures, and a module SERVICE_ROUTINES is offered, too; that includes explicit interface block for all service routines.

A service function can be referenced in a function reference. A service subroutine can be referenced by the service subroutine name in a CALL statement. All service routines are not pure procedures.

The validity of the number and type of arguments is checked if the module SERVICE_ROUTINES that includes all service routines explicit interface block is used. If you use the module SERVICE_ROUTINES in USE statement, specify the ONLY keyword and the service routine name that only using in the program unit, because the module SERVICE_ROUTINES includes all service routines' explicit interface block. The validity of the number and type of arguments is not checked if the module SERVICE_ROUTINES is not used.

Each is documented in detail in the "Chapter 2 Alphabetical Reference". A table is provided in "Appendix B Service Routines".

1.13 Intrinsic Module

The intrinsic modules are provided in the processor. There are a standard intrinsic module and nonstandard intrinsic modules in the intrinsic modules.

1.13.1 Standard Intrinsic Module

The module for interoperable with C language and Fortran language (see "2.288 ISO_C_BINDING Intrinsic Module") , the module for exception process and IEEE Arithmetic (see "1.15 IEEE Exceptions and IEEE Arithmetic") , and Fortran environment module (ISO_FORTRAN_ENV) are the standard intrinsic module (see "2.289 ISO_FORTRAN_ENV Intrinsic Module").

The function defined in the standard intrinsic module is intrinsic module function. The subroutine defined in the standard intrinsic module is intrinsic module subroutine.

1.13.2 Nonstandard Intrinsic Module

The SERVICE_ROUTINES and OMP_LIB modules are the nonstandard intrinsic modules (see "2.415 SERVICE_ROUTINES Nonstandard Intrinsic Module" or "2.353 OMP_LIB Nonstandard Intrinsic Module").

1.14 Scope

Names of program units, common blocks, and external procedures are global entities of a program. That is, they may be referenced from anywhere in the program. A global name shall not identify more than one global entity in a program.

The following name is construct entity and the name has the scope of the construct. Another same identifier name cannot appear in the construct.

- Index-name in a FORALL construct and FORALL statement, or
- Index-name in a DO CONCURRENT (see "2.114 DO Construct") , or
- Associate name in a SELECT TYPE construct, or
- Associate name in an ASSOCIATE construct, or
- An entity that is explicitly declared in the specification part of a BLOCK construct, other than only in ASYNCHRONOUS and VOLATILE statements, is a construct entity.

The name of a variable that appears as an *index-name* in a FORALL construct or DO CONCURRENT has a scope of construct. Within the scope of a construct entity, another construct entity shall not have the same name.

The name of a variable that appears as an *index-name* in a FORALL statement has a scope of statement. The name of a variable that appears as the DO variable of an implied-DO in a DATA statement or an array constructor has a scope of the implied-DO list. The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. Within the scope of a statement entity, another statement entity shall not have the same name.

Other names are local entities of a scoping unit. A scoping unit, is one of the following:

- a derived-type definition

- a procedure interface body, excluding any derived-type definitions or procedure interface bodies in it
- a program unit or subprogram, excluding derived-type definitions, procedure interface bodies, and subprograms in it

That binding label is not case-sensitive and cannot be the same as the binding label of any other language element in the program.

A type attribute specification has the same scope as the derived type definition that contains it. Outside that derived type definition, the type parameter name can be written only as a type parameter keyword within the derived type specifier corresponding to that type, or as a type parameter name of a type parameter query.

The binding name of a type bound procedure has the same scope as the derived type definition that contains it. Outside that derived type definition, the binding name can be written only as a binding name within a procedure reference.

A component name or a binding name can be written only in the scope that can be referenced. If a generic binding has a generic specifier that is not a generic name, it has the same scope as the elements of that type.

The associate name of a SELECT TYPE or ASSOCIATE construct has a separate scope for each block of the construct.

They have the declared type, dynamic type, type parameters, rank and bounds.

If another identifier in the scoping unit of SELECT TYPE or ASSOCIATE construct is the same as an associate name, the name is interpreted within the blocks of the SELECT TYPE or ASSOCIATE construct as that of the associate name.

Elsewhere in the scoping unit, the name is interpreted as another identifier.

1.14.1 Association

To make an entity available to more than one program unit, pass it as an argument (see "1.12.6 Procedure Reference"), place it in a common block (see "2.75 COMMON Statement"), declare it in a module and use the module (see "1.11.2.2 Using Modules"), or host association, described bellow.

1.14.1.1 Host Association

Names in a scoping unit may be referenced from a scoping unit, derived-type definition, subprogram contained within it, and named entities in host scoping unit by an IMPORT statement in the interface body, except when the same name is declared in the inner. This is called a host association.

Even if the host scoping unit elements do not have the VOLATILE attribute or the ASYNCHRONOUS attribute, the referenced elements can have these.

For example,

```
subroutine external ()
  implicit none
  integer :: a, b
  ...
  contains
  subroutine internal ()
    implicit none
    integer :: a
    ...
    a=b  ! a is the local a in internal;
         ! b is available by host association
    ...
  end subroutine internal
end subroutine external
```

In the statement `a=b`, above, `a` is the `a` declared in subroutine `internal`, not the `a` declared in subroutine `external`. `b` is available from `external` by host association.

1.14.1.2 Construct Association

The SELECT TYPE statement (see "2.413 SELECT TYPE Construct") defines the selectors, the associate names, and the associations of that construct. Execution of the ASSOCIATE statement defines the associations between the various selectors and the corresponding associate names of that construct.

If the selector is an allocate entity, it must already be allocated. The associate name associates a data object and does not have the ALLOCATABLE attribute.

If the selector has the POINTER attribute, it must be associated. The associate name associates a pointer target and does not have the POINTER attribute.

If a selector is a variable other than an array section with a vector subscript, the data object specified by the selector is associated. In other cases, the value of the selected expression is associated. The expression is evaluated just before the block is executed. The associate names are associated with the corresponding selector during block execution. These selectors are identified within the block by the corresponding associate name, enabling them to be referenced.

1.15 IEEE Exceptions and IEEE Arithmetic

The intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES provide IEEE exceptions and IEEE arithmetic.

The IEEE_ARITHMETIC module includes the IEEE_EXCEPTIONS use associations. The public items in IEEE_EXCEPTIONS are all public in IEEE_ARITHMETIC.

The types and procedures with use association in these modules are not intrinsic types and intrinsic procedures.

Example:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
USE, INTRINSIC :: IEEE_FEATURES
```

1.15.1 IEEE Derived Types and Constants

The modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES define five derived types with private components.

1.15.1.1 IEEE_EXCEPTIONS

The IEEE_EXCEPTIONS module defines the following:

- IEEE_FLAG_TYPE

  This is used to identify the flag indicating an exception.

  - IEEE_INVALID
  - IEEE_OVERFLOW
  - IEEE_DIVIDE_BY_ZERO
  - IEEE_UNDERFLOW
  - IEEE_INEXACT

  This module defines the following already named array constants:

  - IEEE_USUAL = (/ IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_INVALID /)
  - IEEE_ALL = (/ IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT/)

- IEEE_STATUS_TYPE

  Saves the current floating-point number status.

1.15.1.2 IEEE_ARITHMETIC

The IEEE_ARITHMETIC module defines the following:

- IEEE_CLASS_TYPE

  This is used to identify the type of a floating-point number value.

  - IEEE_SIGNALING_NAN

        - IEEE_QUIET_NAN
        - IEEE_NEGATIVE_INF
        - IEEE_NEGATIVE_NORMAL
        - IEEE_NEGATIVE_DENORMAL
        - IEEE_NEGATIVE_ZERO
        - IEEE_POSITIVE_ZERO
        - IEEE_POSITIVE_DENORMAL
        - IEEE_POSITIVE_NORMAL
        - IEEE_POSITIVE_INF
        - IEEE_OTHER_VALUE
    - IEEE_ROUND_TYPE

      This is used to identify a rounding mode.

        - IEEE_NEAREST
        - IEEE_TO_ZERO
        - IEEE_UP
        - IEEE_DOWN
        - IEEE_OTHER
    - Elemental processing operator ==

      Returns true if the two values are the same, and false if they are not.
    - Elemental processing operator /=

      Returns true if the two values are different, and false if they are not.

1.15.1.3 IEEE_FEATURES

The IEEE_FEATURES module defines the follows:

- IEEE_FEATURES_TYPE

  Shows what is required by IEEE features.

    - IEEE_DATATYPE
    - IEEE_DENORMAL
    - IEEE_DIVIDE
    - IEEE_HALTING
    - IEEE_INEXACT_FLAG
    - IEEE_INF
    - IEEE_INVALID_FLAG
    - IEEE_NAN
    - IEEE_ROUNDING
    - IEEE_SQRT
    - IEEE_UNDERFLOW_FLAG

1.15.2 IEEE Exceptions

The IEEE exceptions are as follows:

- IEEE_OVERFLOW

  If the results of real number intrinsic operations or intrinsic assignments have absolute values that exceed the limits, or if the real or imaginary part of complex number intrinsic operations or intrinsic assignments have absolute values that exceed the limits

- IEEE_DIVIDE_BY_ZERO

  If the dividend of a real number or complex number is other than zero and the divisor is zero

- IEEE_INVALID

  If the operation or assignment of a real number or complex number is invalid

- IEEE_UNDERFLOW

  If it is detected that the precision of a real number intrinsic operation or intrinsic association result has dropped to an absolute value that is lower than the limit, or if it is detected that the precision of the real or imaginary part of a complex number intrinsic operation or intrinsic association has dropped to an absolute value that is lower than the limit

- IEEE_INEXACT

  If the operation or assignment of a REAL type or COMPLEX type is inexact

These exceptions have corresponding flags, taking either exception notification or exception non-notification as their value. The value can be fetched by calling the function IEEE_GET_FLAG. In all cases the initial value is exception non-notification and this changes to exception notification when the corresponding exception occurs. The flag status can be changed using either the IEEE_SET_FLAG or the IEEE_SET_STATUS subroutine. Once the status has changed to exception notification during a procedure, the status remains as exception notification until exception non-notification is set by either the IEEE_SET_FLAG or the IEEE_SET_STATUS subroutine.

An exception may occur, depending on the specification expression evaluation.

1.15.3 IEEE Rounding Mode

There are the following four rounding modes:

- IEEE_NEAREST

  Rounding an exact value to the nearest value that can be represented (nearest rounding)

- IEEE_TO_ZERO

  Rounding an exact value towards zero to the nearest value that can be represented (truncation)

- IEEE_UP

  Rounding an exact value towards + to the nearest value that can be represented (rounding up)

- IEEE_DOWN

  Rounding an exact value towards - to the nearest value that can be represented (rounding down)

The IEEE_GET_ROUNDING_MODE function can be used to query the current mode. The result value is one of the above four modes. If the rounding mode does not conform, the result is IEEE_OTHER.

Use the IEEE_SET_ROUNDING_MODE subroutine to change the rounding mode.

All literal constants with the same format within one program have the same value. Therefore, literal constant values are not affected by the rounding mode.

1.15.4 IEEE Underflow Mode

The underflow mode is controlled by calling the IEEE_SET_UNDERFLOW_MODE function. The IEEE_GET_UNDERFLOW_MODE function can be used to identify which underflow mode is enabled. The IEEE_SUPPORT_UNDERFLOW_MODE function can be used to find out whether or not this functionality is provided.

The underflow mode affects only the type of floating-point number operations for which the IEEE_SUPPORT_UNDERFLOW_CONTROL function returns true.

1.15.5 IEEE Halting Mode

The behavior of a program after an exception occurs (that is, whether execution continues or is halted) can be controlled at execution time using the IEEE_SET_HALTING_MODE subroutine. At the point in time of return, it is guaranteed that the program is at the same point in time at which the halting mode began.

1.15.6 IEEE Floating-Point Status

The values for all flags prepared for each exception, rounding mode, underflow mode, and halting are called as floating-point statuses. The IEEE_GET_STATUS subroutine can be used to save the floating-point status for scalar variables of type TYPE(IEEE_STATUS_TYPE), and the IEEE_SET_STATUS subroutine can be used to set the floating-point status. There is no feature for finding a specific flag value amongst those variables. The IEEE_GET_FLAG, IEEE_GET_HALTING_MODE, and IEEE_GET_ROUNDING_MODE subroutines can be used to save some of the floating-point statuses, and the IEEE_SET_FLAG, IEEE_SET_HALTING_MODE, and IEEE_SET_ROUNDING_MODE subroutines can be used to restore them.

1.15.7 IEEE Exceptional Values

The exceptional floating-point number values are as follows:

- A denormal number is an absolute value that is an extremely small number and has low precision.
- Infinity (+ and -) are generated by overflow or division by zero.
- Not-a-number (NaN) is an undefined value or a value produced as a result of an invalid operation.

Values other than these exceptional values are referred to as normal values.

The functions IEEE_IS_FINITE, IEEE_IS_NAN, IEEE_IS_NEGATIVE, and IEEE_IS_NORMAL are used to check if these values are finite, NaN, negative, or normal. The function IEEE_VALUE generates these types of numbers. These include infinity and NaN. In addition, the functions IEEE_SUPPORT_DENORMAL, IEEE_SUPPORT_INF, and IEEE_SUPPORT_NAN can be used to query whether or not these features can be used for specific types of REAL types.

1.15.8 IEEE Arithmetic

- IEEE_SUPPORT_DATATYPE

  The IEEE_SUPPORT_DATATYPE function can be used to determine whether or not IEEE arithmetic is provided for each REAL type.

- IEEE_SUPPORT_DIVIDE

  The IEEE_SUPPORT_DIVIDE inquiry function can query the division precision.

- IEEE_SUPPORT_NAN

  The IEEE_SUPPORT_NAN inquiry function can be used to determine whether or not IEEE NaN is provided.

- IEEE_SUPPORT_INF

  The IEEE_SUPPORT_INF inquiry function can be used to determine whether or not IEEE infinity is provided.

- IEEE_SUPPORT_DENORMAL

  The IEEE_SUPPORT_DENORMAL inquiry function can be used to determine whether or not IEEE denormal numbers are provided.

- Square root function

  The IEEE_SUPPORT_SQRT function can be used to determine whether or not SQRT for specific types of real numbers is implemented in accordance with the standards.

- IEEE_SUPPORT_STANDARD

  The IEEE_SUPPORT_STANDARD inquiry function can be used to determine whether or not all IEEE functions are provided for the various REAL types.

1.15.9 Overview of IEEE Procedures

For all procedures defined within each module, the displayed dummy argument names are the names used as keyword arguments when specifying actual keyword arguments.

1.15.9.1 Inquiry Functions

- IEEE_EXCEPTIONS

  The IEEE_EXCEPTIONS module has the following inquiry functions:

  - IEEE_SUPPORT_FLAG (see "2.249 IEEE_SUPPORT_FLAG Intrinsic Module Function")

    Are exceptions prepared?

  - IEEE_SUPPORT_HALTING (see "2.250 IEEE_SUPPORT_HALTING Intrinsic Module Function")

    Are IEEE halting controls prepared?

- IEEE_ARITHMETIC

  The IEEE_ARITHMETIC module has the following inquiry functions:

  - IEEE_SUPPORT_DATATYPE (see "2.246 IEEE_SUPPORT_DATATYPE Intrinsic Module Function")

    Is IEEE arithmetic prepared?

  - IEEE_SUPPORT_DENORMAL (see "2.247 IEEE_SUPPORT_DENORMAL Intrinsic Module Function")

    Are IEEE denormal numbers prepared?

  - IEEE_SUPPORT_DIVIDE (see "2.248 IEEE_SUPPORT_DIVIDE Intrinsic Module Function")

    Is IEEE division prepared?

  - IEEE_SUPPORT_INF (see "2.251 IEEE_SUPPORT_INF Intrinsic Module Function")

    Is IEEE infinity prepared?

  - IEEE_SUPPORT_IO (see "2.252 IEEE_SUPPORT_IO Intrinsic Module Function")

    Is IEEE format processing prepared?

  - IEEE_SUPPORT_NAN (see "2.253 IEEE_SUPPORT_NAN Intrinsic Module Function")

    Is IEEE NaN prepared?

  - IEEE_SUPPORT_ROUNDING (see "2.254 IEEE_SUPPORT_ROUNDING Intrinsic Module Function")

    Is the IEEE rounding mode prepared?

  - IEEE_SUPPORT_SQRT (see "2.255 IEEE_SUPPORT_SQRT Intrinsic Module Function")

    Is IEEE square root prepared?

  - IEEE_SUPPORT_STANDARD (see "2.256 IEEE_SUPPORT_STANDARD Intrinsic Module Function")

    Are all IEEE functionalities prepared?

  - IEEE_SUPPORT_UNDERFLOW_CONTROL (see "2.257 IEEE_SUPPORT_UNDERFLOW_CONTROL Intrinsic Module Function")

    Is IEEE underflow control prepared?

1.15.9.2 Elemental Functions

The IEEE_ARITHMETIC module has the elemental functions below. However, these are applied to real number X and Y for which both IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_DATATYPE(Y) are true.

- IEEE_CLASS (see "2.224 IEEE_CLASS Intrinsic Module Function")

  Determines the IEEE real number type.

- IEEE_COPY_SIGN (see "2.225 IEEE_COPY_SIGN Intrinsic Module Function")

  This is the IEEE function copysign.

- IEEE_IS_FINITE (see "2.231 IEEE_IS_FINITE Intrinsic Module Function")

  Determines whether or not the value is finite.

- IEEE_IS_NAN (see "2.232 IEEE_IS_NAN Intrinsic Module Function")

  Determines whether or not the value is IEEE NaN.

- IEEE_IS_NORMAL (see "2.234 IEEE_IS_NORMAL Intrinsic Module Function")

  Determines whether or not the value is normal, that is, whether or not the value is infinity, NaN, or a denormal number.

- IEEE_IS_NEGATIVE (see "2.233 IEEE_IS_NEGATIVE Intrinsic Module Function")

  Determines whether or not the value is negative.

- IEEE_LOGB (see "2.235 IEEE_LOGB Intrinsic Module Function")

  Returns the exponent with the IEEE floating-point padded part removed.

- IEEE_NEXT_AFTER (see "2.236 IEEE_NEXT_AFTER Intrinsic Module Function")

  Returns the number that can be represented that is next to the X in the Y direction.

- IEEE_REM (see "2.237 IEEE_REM Intrinsic Module Function")

  This is the IEEE function rem. That is, it returns X-Y*N. However, N is the integer that is nearest to the X/Y exact value.

- IEEE_RINT (see "2.238 IEEE_RINT Intrinsic Module Function")

  Converts to an integer in accordance with the current rounding mode.

- IEEE_SCALB (see "2.239 IEEE_SCALB Intrinsic Module Function")

  Returns X x 2I.

- IEEE_UNORDERED (see "2.258 IEEE_UNORDERED Intrinsic Module Function")

  This is the IEEE function unordered. This is true when X or Y is NaN. Otherwise it is false.

- IEEE_VALUE (see "2.259 IEEE_VALUE Intrinsic Module Function")

  Generates an IEEE floating-point number.

1.15.9.3 Transformational Function

The IEEE_ARITHMETIC module has the following transformational functions:

- IEEE_SELECTED_REAL_KIND (see "2.240 IEEE_SELECTED_REAL_KIND Intrinsic Module Function")

  Kind type parameter value of the IEEE real number given by the precision and exponent.

1.15.9.4 Elemental Subroutines

The IEEE_EXCEPTIONS module has the following elemental subroutines:

- IEEE_GET_FLAG (see "2.226 IEEE_GET_FLAG Intrinsic Module Subroutine")

  Gets the current exception flag.

- IEEE_GET_HALTING_MODE (see "2.227 IEEE_GET_HALTING_MODE Intrinsic Module Subroutine")

  Gets the halting mode for the exception.

1.15.9.5 Non-elemental Subroutines

- IEEE_EXCEPTIONS

  The IEEE_EXCEPTIONS module has the following non-elemental subroutines:

- IEEE_GET_STATUS (see "2.229 IEEE_GET_STATUS Intrinsic Module Subroutine")

  Gets the current floating-point number environment status.

- IEEE_SET_FLAG (see "2.241 IEEE_SET_FLAG Intrinsic Module Subroutine")

  Sets the exception flag.

- IEEE_SET_HALTING_MODE (see "2.242 IEEE_SET_HALTING_MODE Intrinsic Module Subroutine")

  Controls whether to continue or halt when an exception occurs.

- IEEE_SET_STATUS (see "2.244 IEEE_SET_STATUS Intrinsic Module Subroutine")

  Restores the floating-point number environment status.

The non-elemental subroutines IEEE_SET_FLAG and IEEE_SET_HALTING_MODE are pure. None of the other non-elemental subroutines included in IEEE_EXCEPTIONS are pure.

- IEEE_ARITHMETIC

  The IEEE_ARITHMETIC module has the following non-elemental subroutines:

  - IEEE_GET_ROUNDING_MODE (see "2.228 IEEE_GET_ROUNDING_MODE Intrinsic Module Subroutine")

    Gets the current IEEE rounding mode.

  - IEEE_GET_UNDERFLOW_MODE (see "2.230 IEEE_GET_UNDERFLOW_MODE Intrinsic Module Subroutine")

    Gets the current underflow mode.

  - IEEE_SET_ROUNDING_MODE (see "2.243 IEEE_SET_ROUNDING_MODE Intrinsic Module Subroutine")

    Sets the IEEE rounding mode.

  - IEEE_SET_UNDERFLOW_MODE (see "2.245 IEEE_SET_UNDERFLOW_MODE Intrinsic Module Subroutine")

    Sets the underflow mode.

None of the non-elemental subroutines included in IEEE_ARITHMETIC are pure.

1.16 FORALL Header

FORALL header uses in FORALL construct (see "2.176 FORALL Construct"), FORALL statement (see "2.178 FORALL Statement"), and DO CONCURRENT (see "2.114 DO Construct").

FORALL header is

```
( forall-triplet-spec-list [ , scalar-mask-expr ] )
```

forall-triplet-spec-list is a comma-separated list of

```
index-name = subscript : subscript [ : stride ]
```

index-name is a named scalar variable of type INTEGER.

subscript is a scalar INTEGER expression.

stride is a scalar INTEGER expression.

A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

scalar-mask-expr is a scalar expression of type LOGICAL.

Any procedure referenced in the *scalar-mask-expr* shall be a pure procedure.

1.16.1 Determination of the Value for Index Variables

The subscript and stride expressions in the *forall-triplet-spec* are evaluated. These expressions may be evaluated in any order, and the set of values for index name variable are determined as follows:

1. The subscript *m1*, the subscript *m2*, and the stride *m3* are of type integer with the same kind type parameter as the index name. Their values are results of evaluating the first subscript, the second subscript, and the stride expressions. If a stride does not appear, *m3* has the value 1. The value *m3* shall not be zero.

2. Let the value of *count* be (*m2* - *m1* + *m3*)/ *m3* . If *count* is zero or less for some index name, the execution of the construct is complete. Otherwise, the set of values for the index name is

```
m1+(k-1)* m3          where k = 1,2, ..., count
```

1.16.2 FORALL Mask Expression

If the scalar mask expr appears, the scalar mask expr is evaluated for each combination of index name value. If the scalar mask expr is not present, it is present with the value true.

1.17 Coarray

A coarray is a data entity that has nonzero corank and it can be directly referenced or defined by any image. It may be a scalar or an array. A coarray is specified by ALLOCATABLE statement ("2.15 ALLOCATABLE Statement"), CODIMENSION statement ("2.73 CODIMENSION Statement"), TARGET statement ("2.458 TARGET Statement") ,or type declaration statement ("2.469 Type Declaration Statement") . A coarray must not be a function result. A coarray must not be declared in BLOCK construct ("2.50 BLOCK Construct").

A coarray is an explicit coshape coarray ("1.17.2 Explicit Coshape Coarray") or allocatable coarray ("1.17.3 Allocatable Coarray").

For each coarray on an image, there is a corresponding coarray with the same type, type parameters, and bounds on every other image. A coarray on any image can be accessed directly by using cosubscripts. On its own image, a coarray can also be accessed without use of cosubscripts.

A coarray must be a dummy argument or must have ALLOCATABLE or SAVE attribute.

Coarray specifier must not be specified in data component definition statement of derived type definition (see "1.5.11.1 Derived Type Definition").

A coarray must satisfy following conditions:

- It must not be of type C_PTR or type C_FUNPTR.
- It must not have PARAMETER, POINTER, BIND, or VALUE attribute.
- It must not have VOLATILE attribute if it is referenced by use association or host association.
- It must not appear as an equivalence object or a common block object.
- It must not have a component with ALLOCATABLE or POINTER attribute.
- Its type must not have a component of parameterized derived type or type that have a component of parameterized derived type.

1.17.1 Coarray Specifier

Coarray specifier is an *explicit-coshape-spec* or a *deferred-coshape-spec.* See "1.17.2 Explicit Coshape Coarray" for *explicit-coshape-spec.* See "1.17.3 Allocatable Coarray" for *deferred-coshape-spec.*

1.17.2 Explicit Coshape Coarray

An explicit coshape coarray declares corank and cobounds by an *explicit-coshape-spec.* A nonallocatable coarray must have a coarray specifier that is an *explicit-coshape-spec.*

An *explicit-coshape-spec* is following syntax:

```
[ [ lower-cobound : ] upper-cobound, ] ... [ lower-cobound : ]  *
```

Where:

lower-cobound is specification expression.

upper-cobound is specification expression.

The corank is equal to one plus the number of *upper-cobound*. The sum of the rank and corank of an entity must not exceed 30.

The values of each *lower-cobound* and *upper-cobound* determines the cobounds of the coarray.

If the *lower-cobound* is omitted, the default value is 1. The *upper-cobound* must not be less than the *lower -cobound*.

An explicit coshape coarray must be a dummy argument or have the SAVE attribute.

Example:

```
integer,save:: coarray[2:3,*]
```

1.17.3 Allocatable Coarray

An allocatable coarray is a coarray with ALLOCATABLE attribute. The allocatable coarray declares corank by *deferred-coshape-spec-list*. The cobounds are determined by allocation or argument association.

The *deferred-coshape-spec* is the following syntax:

```
:
```

A coarray with the ALLOCATABLE attribute must have a *coarray-spec* that is a *deferred-coshape-spec-list*.

The corank of an allocatable coarray is equal to the number of colons in its *deferred-coshape-spec-list*. The sum of the rank and corank of an entity must not exceed 30.

The cobounds of an unallocated allocatable coarray are undefined and part of such a coarray must not be referenced or defined.

See ALLOCATE statement ("2.16 ALLOCATE Statement") for allocation of allocatable coarray.

Example:

```
integer,allocatable:: coarray(:) [:,:]
```

1.17.4 Coarray Reference

Syntax:

```
part [ %part ] ...
```

Where:

part is following syntax:

```
part-name [ ( subscript-list ) ] [ image-selector ] [ ( substrings ) ]
```

part-name of the leftmost *part* must be a coarray.

Using '%' operator is structure component reference (see "1.5.11.6 Structure Component").

If (*subscript-list*) is specified, *name* must be an array. See "1.5.8.1 Array References" for reference of arrays.

An entity with an *image-selector* (see "1.17.5 Image Selector") is a coindexed object. A coindexed object must satisfy following conditions:

- (*subscript-list*) must be specified if part name is an array.
- *subscript* must not be a vector subscript.
- It must not be of type C_PTR or C_FUNPTR.
- Part name must not be a selector.
- It must not be a pointer component or allocatable component.

Image selector can be omitted on its own image.

A subobject of a coarray is coarray, except that coindexed object is not a coarray.

If (*substrings*) is specified, *name* must be of type character. See "1.5.7 Substrings" for reference of substring.

1.17.5 Image Selector

An image selector determines the image index for a coindexed object.

Syntax:

left-square-bracket cosubscript-list right-square-bracket

Where:

cosubscript is a scalar integer expression.

left-square-bracket is a left-side square bracket. The left-side square bracket is '['.

right-square-bracket is a right-side square bracket. The right-side square bracket is ']'.

The number of *cosubscripts* must be equal to the corank of the object. The value of a *cosubscript* in an image selector must be within the cobounds of its codimension. Taking account of the cobounds, the cosubscript list in an image selector determines the image index in the same way that a subscript list in an array element determines the subscript order value, taking account of the bounds.

If image selector appears, data reference must not be of type C_PTR or C_FUNPTR.

A coindexed object must not be polymorphic.

Example:

```
integer,save:: coarray_scalar [ * ]
integer,save:: coarray_array(2) [ 2 , * ]
coarray_scalar[ 1 ] = 1
coarray_array( 2 ) [ 1, 1 ] = 2
coarray_array( 1:2 ) [ 1, 1 ] = 3
coarray_array( : ) [ 1, 1 ] = 4

coarray_array [ 1, 1 ] = 5  !  error. ( subscript-list ) must appear.
```

1.18 Image Control Statement

The execution order on an image is divided into segments by execution of image control statements. Followings are image control statements:

- SYNC ALL statement (see "2.449 SYNC ALL Statement").
- SYNC IMAGES statement (see "2.450 SYNC IMAGES Statement").
- SYNC MEMORY statement (see "2.451 SYNC MEMORY Statement").
- ALLOCATE statement (see "2.16 ALLOCATE Statement") or DEALLOCATE statement (see "2.110 DEALLOCATE Statement") with a coarray allocate object.
- CRITICAL statement (see "2.96 CRITICAL Construct") or END CRITICAL statement (see "2.131 END CRITICAL Statement").
- LOCK statement (see "2.311 LOCK Statement") or UNLOCK statement (see "2.477 UNLOCK Statement").
- Any statement that completes a block or procedure, and deallocates a coarray implicitly.
- MOVE_ALLOC intrinsic subroutine (see "2.341 MOVE_ALLOC Intrinsic Subroutine") with coarray dummy arguments.
- STOP statement (see "2.443 STOP Statement").
- END statement (see "2.127 END Statement") of main program.

All image control statements except CRITICAL, END CRITICAL, LOCK and UNLOCK statements include the effect of SYNC MEMORY statement.

A statement must not cause more than one image statement executed, except CRITICAL and END CRITICAL statements.

1.18.1 Segment

Segment is a sequence of statements divided by image control statements on each image. A segment immediately precedes an image control statement includes evaluations of all expressions within the image control statement.

The execution ordering of i-th segment P_i on image P and j-th segment Q_j on image Q can be ensured by execution of image control statement or user-defined ordering. See "2.451 SYNC MEMORY Statement" for user-defined ordering. If the ordering is not ensured, P_i and Q_j may be executed at the same time depending on the execution speeds of the images. Segments executed by a single image are totally ordered.

A coarray can be defined or referenced by an atomic subroutine even if segments are not ordered. If atomic subroutine is not used, following condition must be satisfied:

- If a variable is defined in a segment on an image, the variable must not be referenced, defined, or become undefined on another segment unless these segments are ordered.
- If allocation of allocatable component of a coarray or pointer association status of a pointer component of a coarray is changed on a segment on an image, the component must not be defined or referenced on another segment unless these segments are ordered.
- If an execution of a procedure that defines noncoarray dummy arguments continues during segments P_i, P_i+1, ... , P_k, the effective argument must not be referenced, defined, or become undefined in segment Q_j on image Q unless Q_j precedes P_i or succeeds P_k.

1.18.2 Synchronization Status Specifier

Synchronization status specifier includes two specifiers: STAT specifier for status variable, and ERRMSG specifier for error message variable. See "2.289 ISO_FORTRAN_ENV Intrinsic Module" for the named constants assigned to status variable.

If execution of a LOCK, SYNC ALL, SYNC IMAGES, SYNC MEMORY, or UNLOCK statement succeeds, status variable is defined by the value 0.

If a SYNC ALL or SYNC IMAGES statement that includes STAT specifier causes synchronization with an image that initiates termination, status variable is defined with the value of STAT_STOPPED_IMAGE. Then, the SYNC ALL or SYNC IMAGE statement has the same effect as SYNC MEMORY statement.

If the STAT specifier appears in a LOCK statement and the lock variable is locked by the executing image, the status variable is defined with the value of STAT_LOCKED. If the STAT specifier appears in an UNLOCK statement and the lock variable is not locked, status variable is defined with the value of STAT_UNLOCKED. If the STAT specifier appears in an UNLOCK statement and the lock variable is locked by another image, status variable is defined with the value of STAT_LOCKED_OTHER_IMAGE.

If an error condition occurs during execution of a LOCK, SYNC ALL, SYNC IMAGES, SYNC MEMORY, or UNLOCK statement without STAT specifier, error termination begins.

If an error condition occurs during execution of a LOCK, SYNC ALL, SYNC IMAGES, SYNC MEMORY, or UNLOCK statement with ERRMSG specifier, an explanation message of the error is assigned to error message variable. If no error condition occurs, the value of the variable is not changed.

1.18.3 LOCK_TYPE Type

LOCK_TYPE is a derived type with private components. No components is allocatable or a pointer. It is extendable type with no type parameter. It does not have BIND (C) attribute and is not a sequence type. All components have default initialization.

A scalar variable of type LOCK_TYPE is a lock variable. Lock variables have two states: locked or unlocked. A lock variable is unlocked if it has initial value of a LOCK_TYPE variable; it is the value of LOCK_TYPE(). Otherwise it is locked. The values of lock variables are changed by LOCK and UNLOCK statements.

A named variable of type LOCK_TYPE must be a coarray. A named variable that have a noncoarray component of type LOCK_TYPE must be a coarray.

A lock variable must not appear in a variable definition context except following cases:

- as a lock variable in a LOCK or UNLOCK statement,
- as an allocate object in an ALLOCATE or DEALLOCATE statement, or

- as an actual argument in a reference of a procedure with an explicit interface, and the corresponding dummy argument has an INTENT(INOUT) attribute.

A variable with a component of LOCK_TYPE must not appear in a variable definition context except following cases:

- as an allocate object in an ALLOCATE or DEALLOCATE statement, or
- as an actual argument in a reference of a procedure with an explicit interface, and the corresponding dummy argument has an INTENT(INOUT) attribute.

If EXTENDS appears to a derived type definition (see "1.5.11.1 Derived Type Definition") and the type being defined has an ultimate component of type LOCK_TYPE, its parent type must have an ultimate component of type LOCK_TYPE.

Chapter 2 Alphabetical Reference

2.1 ABORT Service Subroutine

Description

ABORT writes a message to the standard error file and abnormally ends the execution of the program.

Syntax

CALL ABORT

Example

```
      use service_routines,only:abort
      open(10,file='x.dat',err=10)
      call sub
      stop 'program main end'
   10 call abort
      end
      subroutine sub
      write(6,fmt="(1x,a)") "abort test"
      end
```

2.2 ABS Intrinsic Function

Description

Absolute value.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ABS	---	1	One-byte INTEGER	One-byte INTEGER
	I2ABS		Two-byte INTEGER	Two-byte INTEGER
	IIABS		Two-byte INTEGER	Two-byte INTEGER
	IABS		Four-byte INTEGER	Four-byte INTEGER
	JIABS		Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER	Eight-byte INTEGER
	ABS		Single-prec REAL	Single-prec REAL
	DABS		Double-prec REAL	Double-prec REAL
	QABS		Quad-prec REAL	Quad-prec REAL

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	CABS		Single-prec COMPLEX	Single-prec REAL
	CDABS		Double-prec COMPLEX	Double-prec REAL
	CQABS		Quad-prec COMPLEX	Quad-prec REAL

```
result = ABS ( A )
```

Argument(s)

A

A shall be of type REAL, INTEGER, or COMPLEX.

Result

If *A* is of type INTEGER or REAL, the result is of the same type as *A* and has the value |*A*| ; if *A* is COMPLEX with value (*x*,*y*), the result is a REAL representation of $\sqrt{(x^2+y^2)}$.

Remarks

ABS, I2ABS, IIABS, IABS, JIABS, DABS, QABS, CABS, CDABS, and CQABS evaluate the absolute value of integer, real, or complex data.

The generic name, ABS, may be used with any integer, real, or complex argument.

The type of the result of each function is the same as the type of the argument except when the argument is complex the result is real with the same kind type parameter as the argument.

Example

```
x = abs(-4.0)        ! x is assigned the value 4.0
```

2.3 ACCESS Service Function

Description

Determines if a file exists and how it can be accessed.

Syntax

```
iy = ACCESS ( fname , mode )
```

Argument(s)

fname

Default CHARACTER scalar. Name of the file whose accessibility is to be determined.

mode

Default CHARACTER scalar. Mode(s) of accessibility to check for. These characters are interpreted as follows.

```
'r'                     : Tests for read permission
'w'                     : Tests for write permission
'x'                     : Tests for execute permission
' '(A blank character)    : Tests for existence
```

Result

Default INTEGER scalar. The result is -1 if either argument is illegal; otherwise, a system error code is returned.

Example

```
!  'test.f90' exists in the current directory and its access permission
!  is 'rw'.
use service_routines,only:access
write(6,*) access("test.f90", "r")    ! Returns 0
write(6,*) access("test.f90", "x")    ! Returns the system error code.
write(6,*) access("test.f90", " ")    ! Returns 0
write(6,*) access("test.f90", "rw ")  ! Returns 0
end
```

2.4 ACHAR Intrinsic Function

Description

Character in a specified position of the ASCII collating sequence.

Class

Elemental function.

Syntax

result = ACHAR (*I* [, *KIND*])

Required Argument(s)

I

I shall be of type INTEGER.

Optional Argument(s)

KIND

A scalar INTEGER initialization expression.

Result

A CHARACTER string of length 1. If *KIND* is specified, the kind type parameter matches the *KIND* value. If *KIND* is omitted, the kind type parameter is the default CHARACTER type.

Remarks

The character is returned in position *I* of the ASCII collating sequence.

ACHAR(IACHAR(C)) has the value C for any character C that can be expressed in the default CHARACTER type.

The result is the CHARACTER in position *I* of the ASCII collating sequence with 0 <= *X* <= 255 .

If the kind type parameter *KIND* is specified, the function result type is a CHARACTER of length 1 and has the size *KIND*. If *KIND* is omitted, the function result type is the default CHARACTER type of length 1.

Example

```
c = achar(65)    ! c is assigned the value 'A'
```

2.5 ACOS Intrinsic Function

Description

Arccosine, expressed in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ACOS	---	1	REAL or COMPLEX	REAL or COMPLEX
	ACOS		Single-prec REAL	Single-prec REAL
	DACOS		Double-prec REAL	Double-prec REAL
	QACOS		Quad-prec REAL	Quad-prec REAL

```
result = ACOS ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX. If it is of type REAL, the domain shall be within the range ABS(*X*) <= 1.0 .

Result

A REAL or COMPLEX representation, expressed in radians, of the arccosine of *X*.

Remarks

ACOS evaluates the arccosine, expressed in radians for type REAL or COMPLEX. DACOS and QACOS evaluate the arccosine, expressed in radians for type REAL.

X shall satisfy the inequality ABS(*X*) <= 1.0 .

If the result is of type REAL, the result lies in the range 0 <= ACOS(*X*) <= π . If the result is of type COMPLEX, the real part result as a value in radians lies in the range 0 <= REAL(ACOS(*X*)) <= π .

The generic name, ACOS, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = acos(.5)
```

2.6 ACOSD Intrinsic Function

Description

Arccosine, expressed in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ACOSD	---	1	REAL	REAL
	ACOSD		Single-prec REAL	Single-prec REAL
	DACOSD		Double-prec REAL	Double-prec REAL
	QACOSD		Quad-prec REAL	Quad-prec REAL

```
result = ACOSD ( X )
```

Argument(s)

X

X shall be of type REAL and shall be within the range -1.0 <= *X* <= 1.0 .

Result

A REAL representation, expressed in degrees, of the arccosine of *X*.

Remarks

ACOSD, DACOSD, and QACOSD evaluate the arccosine, expressed in degrees. *X* shall satisfy the inequality ABS(*X*) <= 1.0 . The result has the value 180/ π * ACOS(*X*) and lies in the range 0.0 <= ACOSD(*X*) <= 180.0 .

The generic name, ACOSD, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = acosd(.5)
```

2.7 ACOSH Intrinsic Function

Description

Inverse hyperbolic cosine.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ACOSH	---	1	REAL or COMPLEX	REAL or COMPLEX

```
result = ACOSH ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX. If it is of type REAL, the domain shall be within the range *X* >= 1.0 .

Result

The result has the same kind as *X*.

Remarks

ACOSH evaluates the inverse hyperbolic cosine for type REAL or COMPLEX.

If *X* is of type COMPLEX, its imaginary part is treated as a value in radians, and lies in the range 0 <= AIMAG(ACOSH(*X*)) <= π .

The type of the result is the same as the type of the argument.

Example

```
r = acosh(1.5)
```

2.8 ACOSQ Intrinsic Function

Description

Arccosine, expressed in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ACOSQ	---	1	REAL	REAL
	ACOSQ		Single-prec REAL	Single-prec REAL
	DACOSQ		Double-prec REAL	Double-prec REAL
	QACOSQ		Quad-prec REAL	Quad-prec REAL

```
result = ACOSQ ( X )
```

Argument(s)

X

X shall be of type REAL and shall be within the range -1.0 <= *X* <= 1.0 .

Result

A REAL representation, expressed in quadrants, of the arccosine of *X*.

Remarks

ACOSQ, DACOSQ, and QACOSQ evaluate the arccosine, expressed in quadrants. *X* shall satisfy the inequality ABS(*X*) <= 1.0 . The result has the value 2/ π * ACOS(*X*) and lies in the range 0.0 <= ACOSQ(*X*) <= 2.0 .

The generic name, ACOSQ, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = acosq(.5)
```

2.9 ADJUSTL Intrinsic Function

Description

Adjust to the left, removing leading blanks and inserting trailing blanks.

Class

Elemental function.

Syntax

```
result = ADJUSTL ( STRING )
```

Argument(s)

STRING

STRING shall be of type CHARACTER.

Result

A CHARACTER of the same length and kind as *STRING*. Its value is the same as that of *STRING* except that any leading blanks have been deleted and the same number of trailing blanks has been inserted.

Example

```
character(len=7)::adjusted
adjusted = adjustl(' string')
         ! adjusted is assigned the value 'string '
```

2.10 ADJUSTR Intrinsic Function

Description

Adjust to the right, removing trailing blanks and inserting leading blanks.

Class

Elemental function.

Syntax

```
result = ADJUSTR ( STRING )
```

Argument(s)

STRING

STRING shall be of type CHARACTER.

Result

A CHARACTER of the same length and kind as *STRING*. Its value is the same as that of *STRING* except that any trailing blanks have been deleted and the same number of leading blanks has been inserted.

Example

```
character(len=7)::adjusted
adjusted = adjustr('string ')
         ! adjusted is assigned the value ' string'
```

2.11 AIMAG Intrinsic Function

Description

Imaginary part of a complex number.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
AIMAG or IMAG	---	1	COMPLEX	REAL
	AIMAG		Single-prec COMPLEX	Single-prec REAL
	IMAG		Single-prec COMPLEX	Single-prec REAL
	DIMAG		Double-prec COMPLEX	Double-prec REAL
	QIMAG		Quad-prec COMPLEX	Quad-prec REAL

```
result = AIMAG ( Z )
result = IMAG ( Z )
```

Argument(s)

Z

Z shall be of type COMPLEX.

Result

A REAL with the same kind as *Z*. If *Z* has the value (*x*,*y*) then the result has the value *y*.

Remarks

AIMAG, IMAG, DIMAG, and QIMAG obtain the imaginary part of a complex item.

The generic names, AIMAG and IMAG, may be used with any COMPLEX argument.

The type of the result of each function is REAL with the same kind type parameter as the argument.

Example

```
r = aimag((-4.0,5.0))  ! r is assigned the value 5.0
```

2.12 AINT Intrinsic Function

Description

Truncation to a whole number.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
AINT	---	1 or 2	REAL [, INTEGER]	REAL
	AINT	1	Single-prec REAL	Single-prec REAL
	DINT		Double-prec REAL	Double-prec REAL
	QINT		Quad-prec REAL	Quad-prec REAL

```
result = AINT ( A [ , KIND ] )
```

Required Argument(s)

A

A shall be of type REAL.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

A REAL value with the kind specified by *KIND*, if present; otherwise with the kind of *A*. The result is equal to the value of *A* without its fractional part.

Remarks

AINT, DINT, and QINT truncate to a whole number.

The result is 0.0, if ABS(*A*) < 1.0 ; otherwise, the result 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*.

The generic name, AINT, may be used with any REAL argument.

If the kind type parameter *KIND* is specified, *KIND* type of each function result is that specified by *KIND*. If *KIND* is omitted, the function result type is the same as the type of the argument *A*.

Example

```
r = aint(-7.32,4)  ! r is assigned the value -7.0
                   ! with kind 4
```

2.13 ALARM Service Function

Description

Causes a subroutine to begin execution after a specified amount of time has elapsed.

Syntax

```
iy = ALARM( time , sub )
```

Argument(s)

time

Default INTEGER scalar. Specifies the time delay in seconds.
If *time* is 0, the alarm is turned off and no routine is called.

sub

Name of the procedure to call.

Result

Default INTEGER scalar. Returns the number of seconds remaining until the previously set alarm is to go off.

Example

```
use service_routines,only:alarm
integer :: iy
external :: sub
iy = alarm(2, sub)     ! Executes SUB after 2 seconds.
...
end
```

2.14 ALL Intrinsic Function

Description

Determine whether all values in a mask are true along a given dimension.

Class

Transformational function.

Syntax

```
result = ALL ( MASK [ , DIM ] )
```

Required Argument(s)

MASK

MASK shall be of type LOGICAL. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar of type INTEGER with a value within the range 1 <= *DIM* <= *n* , where *n* is the rank of *MASK*. The corresponding actual argument shall not be an optional dummy argument.

Result

The result is of type LOGICAL with the same kind as *MASK*. Its value and rank are computed as follows:

1. If *DIM* is absent or *MASK* has rank one, the result is scalar. The result has the value true if all elements of *MASK* are true.
2. If *DIM* is present or *MASK* has rank two or greater, the result is an array of rank *n*-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of *MASK* and *n* is the rank of *MASK*. The result has the value true for each corresponding vector in *MASK* that evaluates to true for all elements in that vector.

Example

```
integer, dimension (2,3) :: a, b
logical, dimension (2) :: c
logical, dimension (3) :: d
logical :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                !  represents  |1 3 5|
                !              |2 4 6|
b = reshape((/1,2,3,5,6,4/), (/2,3/))
                !  represents  |1 3 6|
                !              |2 5 4|
e = all(a==b)   ! e is assigned the value false
d = all(a==b, 1)! d is assigned the value true,false,false
c = all(a==b, 2)! c is assigned the value false,false
```

2.15 ALLOCATABLE Statement

Description

The ALLOCATABLE statement declares allocatable variables. If an array is being declared, an allocatable array shall be a deferred-shape array. The shape of an allocatable array is determined when space is allocated for it by an ALLOCATE statement.

Syntax

```
ALLOCATABLE [ :: ] allocatable-decl-list
```

allocatable-decl-list is

```
object-name [ ( deferred-shape-spec-list ) ]       &
&       [ left-square-bracket  coarray-spec  right-square-bracket ]
```

Where:

object-name is the name of an object.

deferred-shape-spec-list is a comma-separated list of

:

coarray-spec is a coarray specifier. See "1.17.1 Coarray Specifier" for coarray specifier.

left-square-bracket is a left-side square bracket. The left-side square bracket is '['.

right-square-bracket is a right-side square bracket. The right-side square bracket is ']'.

Remarks

If the DIMENSION of *array-name* is specified elsewhere in the scoping unit, it shall be specified as a *deferred-shape-spec-list*.

Example

```
integer :: a,b,c(:,:,:)   ! rank of c is specified
dimension :: b(:,:)       ! rank of b is specified
allocatable :: a(:),b,c   ! rank of a is specified,
                          ! a,b, and c are allocatable
allocate (a(2), b(3,-1:1), c(10,10,10))
                          ! shapes specified,
                          ! space allocated
   ...
deallocate (a,b,c)        ! space deallocated
```

2.16 ALLOCATE Statement

Description

The ALLOCATE statement dynamically creates pointer targets and allocatable variables.

Syntax

```
ALLOCATE ( [ type-spec :: ] allocation-list [ , alloc-opt-list ] )
```

Where:

type-spec is a type specifier that is either an intrinsic type specifier or a derived type specifier.

See "2.469 Type Declaration Statement" for intrinsic type specifier, and see "1.5.11.8 Derived Type Specifier" for derived type specifier.

allocation-list is a comma-separated list of

```
allocate-object [ ( allocate-shape-spec-list ) ]      &
&    [ left-square-bracket allocate-coarray-spec right-square-bracket ]
```

allocate-object is

```
variable-name        or
structure-component
```

variable-name is the name of the variable.

structure-component is the structure component.

allocate-object shall be a pointer or an allocatable variable.

allocate-object may be of type character with zero character length.

allocate-object must not be a coindexed object.

allocate-shape-spec-list is a comma-separated list of

```
[ allocate-lower-bound : ] allocate-upper-bound
```

Both *allocate-lower-bound* and *allocate-upper-bound* shall be scalar INTEGER expressions.

allocate-coarray-spec is

```
[ allocate-coshape-spec-list, ]  [ allocate-lower-bound : ] *
```

allocate-coshape-spec is

```
[ allocate-lower-bound : ]  allocate-upper-bound
```

An *allocate-coarray-spec* must appear if and only if the *allocate-object* is a coarray.

The number of *allocate-coshape-spec* in an *allocate-coarray-spec* must be one less than the corank of the *allocate-object*.

type-spec must not specify the type C_PTR or C_FUNPTR if an *allocate-object* is a coarray.

left-square-bracket is a left-side square bracket. The left-side square bracket is '['.

right-square-bracket is a right-side square bracket. The right-side square bracket is ']'.

alloc-opt-list is a list of options for allocation, that is

```
STAT = stat-variable          or
ERRMSG = errmsg-variable      or
SOURCE = source-expr          or
MOLD = source-expr
```

stat-variable is a scalar INTEGER variable.

errmsg-variable is an error notification variable of a scalar default CHARACTER type.

source-expr is a source expression.

The declared type of *source-expr* must not be C_PTR, C_FUNPTR, LOCK_TYPE, or have a subcomponent of type LOCK_TYPE, if an *allocate-object* is a coarray.

Remarks

The number of *allocate-shape-spec* in an *allocate-shape-spec-list* must be the same as the rank of the *allocate-object*.

allocate-shape-spec-list can be omitted, if an array is specified in SOURCE= specifier or MOLD= specifier.

The *stat-variable*, *source-expr*, and *errmsg-variable* cannot be allocated by an ALLOCATE statement in which they appear. In addition, they cannot depend on any values, shape specifications, length type parameters, allocation states, or association states in allocate objects that appear in the same ALLOCATE statement.

If a STAT= specifier ALLOCATE statement has executed normally, the value 0 is set in the *stat-variable*. If an error condition occurred during execution of the ALLOCATE, the execution-time diagnostic message number is set in the *stat-variable*. The status of an allocate object for which allocation has succeeded is either the "allocated" status or the pointer association status "associated". The status of an allocate object for which allocation fails remains the previous allocation status or pointer association status.

If an error condition occurred during execution of the ALLOCATE statement, the status is assigned to the *errmsg-variable*. If no error conditions have occurred, the value of the *errmsg-variable* is not changed.

If an error condition occurs during execution of an ALLOCATE statement that does not contain the STAT= specifier, execution of the executable program is terminated.

At the time an ALLOCATE statement is executed for an array with *allocate-shape-spec-list*, the values of the *allocate-lower-bound* and *allocate-upper-bound* expressions determine the bounds of the array. Subsequent redefinition or undefinition of any entities in the bound expressions does not affect the array bounds. If the *allocate-lower-bound* is omitted, the default value is 1.
If the *allocate-upper-bound* is less than the *allocate-lower-bound*, the extent in that dimension is zero and the array has zero size.

When an ALLOCATE statement is executed for an array with no *allocate-shape-spec-list*, the bounds of *source-expr* determine the bounds of the array.

When an ALLOCATE statement is executed for a coarray, the values of the lower cobound and upper cobound expressions determine the cobounds of the coarray. Subsequent redefinition or undefinition of any entities in the cobound expressions does not affect the cobounds. If the lower cobound is omitted, the default value is 1. The upper cobound must not be less than the lower cobound.

The ALLOCATED intrinsic function (see "2.17 ALLOCATED Intrinsic Function") may be used to determine whether an allocatable variable is currently allocated.

If the length type parameter value of the allocate object is specified and differs from the value of the corresponding type parameter of *source-expr* in SOURCE= specifier or MOLD= specifier, an error condition occurs. If allocation has succeeded, the allocate object value becomes the *source-expr* value in SOURCE= specifier.

When allocate a pointer that is currently associated with a target, a new pointer target is created as required by the attributes of the pointer and any array bounds specified in the ALLOCATE statement.

The ASSOCIATED intrinsic function (see "2.29 ASSOCIATED Intrinsic Function") may be used to determine whether a pointer is currently associated.

At the beginning of execution of a function whose result is a pointer, the association status of the result pointer is undefined. Before such a function returns, it shall either associate a target with this pointer or cause the association status of this pointer to become defined as disassociated.

These allocate objects are either pointers that do not specify procedures, or allocate variables.

If allocate objects in the statement have deferred type parameters, a type specifier, SOURCE= specifier, or MOLD= specifier must be set.

If there is a type specifier, the type being specified must be compatible with the allocate objects and types.

If an allocate object is unlimited polymorphic or an abstract type, a type specifier, SOURCE= specifier, or MOLD= specifier must be set.

Unless the type parameters corresponding to these allocate objects are inherited dummy arguments, the type parameter value in the type specifier must be '*'.

If there is a type specifier, the kind type parameter value of the allocate objects must be the same as the corresponding kind type parameter value in that type specifier.

The allocate shape specification list must be specified if the allocate object is an array and if there is no array *source-expr*. The allocate shape specification list must not be specified if the allocate object is a scalar.

If the allocate object is an array and allocate shape specification list is specified:

- If a SOURCE= specifier is specified, the *source-expr* must be shape compatible with allocate shape specification list.
- If a MOLD= specifier is specified, the *source-expr* need not be same rank with allocate object.

If the allocate object is an array and allocate shape specification list is omitted, the *source-expr* in SOURCE= specifier or MOLD= specifier must be same rank with allocate object.

The number of allocate shape specifications in the allocate shape specification list must be the same as the allocate object rank.

The same allocate selector cannot be specified more than once in one allocate selector list.

If there is a SOURCE= specifier, there cannot be a type specifier or MOLD= specifier. In addition, there must be only one allocate object in the allocate list, and the *source-expr* and type must be compatible.

If there is a MOLD= specifier, there cannot be a type specifier or SOURCE= specifier. In addition, the *source-expr* and an allocate object must be type compatible.

If there is a MOLD= specifier, the *source-expr* must not be of parameterized derived type.

The corresponding allocate object and the *source-expr* kind type parameter must be the same value.

An allocate object, its shape specification parameter, and its type parameter do not depend on the value of the *stat-variable* or the value of the *errmsg-variable*. Nor do they depend on the values, shape specification, length type parameter, allocate status, or associate status in the allocate objects in the same ALLOCATE statement.

If there is a type specifier, when a polymorphic object is allocated, an object with the specified dynamic type is allocated. If there is a *source-expr*, an object with the same dynamic type and type parameter as that expression is allocated. In other cases, an object with a dynamic type that is the same as the declared type is allocated.

When an ALLOCATE statement is executed in accordance with a type specifier, the type parameter value in the type specifier specifies the type parameter. If the value specified as the type parameter differs from the value specified during declaration of the allocate object, an error condition occurs.

If '*' is the type parameter value in the type specifier in the ALLOCATE statement, the current value of the inherited type parameter is indicated. If the type parameter value is an expression, the type parameter value is not affected even if variables in that expression are subsequently redefined or undefined.

If a *source-expr* is a pointer, it must be associated. If a *source-expr* is an allocatable, it must be allocated.

If a MOLD= specifier is specified and the *source-expr* is a variable, its value need not be defined.

Even if the allocate has executed normally when MOLD= specifier is specified, the value of the allocated object is undefined.

If an *allocate-object* is a coarray, the declared type of *source-exp* must not be C_PTR, C_FUNPTR or LOCK_TYPE, or have a subcomponent of type LOCK_TYPE.

If an allocation specifies a coarray, its dynamic type and the values of corresponding type parameters must be the same on every image. The values of corresponding bounds and corresponding cobounds must be the same on every image. If the coarray is a dummy argument, the actual argument must be the same coarray on every image.

When an ALLOCATE statement is executed for which an *allocate-object* is a coarray, there is an implicit synchronization of all images. On each image, execution of the segment following the statement is delayed until all other images have executed the same statement.

Example

```
integer, pointer :: n
integer, allocatable, dimension (:,:) :: k
integer, allocatable, dimension (:) , codimension [:,:] :: ca ! coarray
allocate (k(4,-2:3))                  ! shape of k defined, space allocated
allocate (n)                          ! n points to unnamed target
allocate (ca(3) [ 2,* ] )             ! coshape of ca defined, space allocated
```

2.17 ALLOCATED Intrinsic Function

Description

Indicate whether an allocatable variable has been allocated.

Class

Inquiry function.

Syntax

```
result = ALLOCATED ( ALLOCATABLE )
```

Argument(s)

ALLOCATABLE

ALLOCATABLE shall be an allocatable variable.

Result

The result is a scalar of default LOGICAL type. It has the value true if *ALLOCATABLE* is currently allocated and false if *ALLOCATABLE* is not currently allocated.

Example

```
integer, allocatable :: i(:,:)
logical l
allocate (i(2,3))
l = allocated (i) ! l is assigned the value true
```

2.18 ANINT Intrinsic Function

Description

REAL representation of the nearest whole number.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ANINT	---	1 or 2	REAL [, INTEGER]	REAL
	ANINT	1	Single-prec REAL	Single-prec REAL
	DNINT		Double-prec REAL	Double-prec REAL
	QNINT		Quad-prec REAL	Quad-prec REAL

```
result = ANINT ( A  [ , KIND ] )
```

Required Argument(s)

A

A shall be of type REAL.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is of type REAL. If *KIND* is specified, the kind type parameter of the function result is following to *KIND*. If *KIND* is omitted, the kind type parameter of the function result is the same as *A*.

If *A* > 0, the result has the value AINT(*A* + 0.5); if *A* <= 0, the result has the value AINT(*A* - 0.5).

Remarks

ANINT, DNINT, and QNINT evaluate to the nearest whole number. If ABS(*A*) > 0 , the result has the value AINT(*A* + 0.5). Otherwise, the result has the value AINT(*A* - 0.5).

The generic name, ANINT, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
x = anint (7.73)  ! x is assigned the value 8.0
```

2.19 ANY Intrinsic Function

Description:

Determine whether any values are true in a mask along a given dimension.

Class

Transformational function.

Syntax

```
result = ANY ( MASK [ , DIM ] )
```

Required Argument(s)

MASK

MASK shall be of type LOGICAL. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar of type INTEGER with a value within the range 1 <= *DIM* <= *n* , where *n* is the rank of *MASK*. The corresponding actual argument shall not be an optional dummy argument.

Result

The result is of type LOGICAL with the same kind as *MASK*. Its value and rank are computed as follows:

1. If *DIM* is absent or *MASK* has rank one, the result is scalar. The result has the value true if any elements of *MASK* are true.
2. If *DIM* is present or *MASK* has rank two or greater, the result is an array of rank *n*-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of *MASK* and n is the rank of *MASK*. The result has the value true for each corresponding vector in *MASK* that evaluates to true for any element in that vector.

Example

```
integer, dimension (2,3) :: a, b
logical, dimension (2) :: c
logical, dimension (3) :: d
logical :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                !  represents  |1 3 5|
                !              |2 4 6|
b = reshape((/1,2,3,5,6,4/), (/2,3/))
                !  represents  |1 3 6|
                !              |2 5 4|
e = any(a==b)   ! e is assigned the value true
d = any(a==b, 1)! d is assigned the value true,true,false
c = any(a==b, 2)! c is assigned the value true,true
```

2.20 Arithmetic IF Statement (obsolescent feature)

Description

Execution of arithmetic IF statement causes evaluation of an expression followed by a transfer of control.

Syntax

```
IF ( scalar-numeric-expr ) label , label , label
```

Where:

scalar-numeric-expr is a scalar numeric expression and shall not be of type COMPLEX.

label is a statement label. Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the arithmetic IF statement.

Remarks

The same *label* can appear more than once in one arithmetic IF statement.

Execution of arithmetic IF statement causes evaluation of the *scalar-numeric-expr* followed by a transfer of control. The branch target statement identified by the first, second, or third label is executed next as the value of the expression is less than zero, equal to zero, or greater than zero, respectively.

Example

```
if (b) 10,20,30 ! go to 10 if b<0
                ! go to 20 if b==0
                ! go to 30 if b>0
```

2.21 ASIN Intrinsic Function

Description

Arcsine, expressed in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ASIN	---	1	REAL or COMPLEX	REAL or COMPLEX
	ASIN		Single-prec REAL	Single-prec REAL
	DASIN		Double-prec REAL	Double-prec REAL
	QASIN		Quad-prec REAL	Quad-prec REAL

```
result = ASIN ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX. If it is type REAL, the domain shall be in the range ABS(*X*) <= 1.0 .

Result

The result has the same kind as *X*. Its value is a REAL or COMPLEX representation of the arcsine of *X*, expressed in radians.

Remarks

ASIN evaluates the arcsine, expressed in radians for type REAL or COMPLEX. DASIN and QASIN evaluate the arcsine, expressed in radians for type REAL.

X shall satisfy the inequality ABS(*X*) <= 1.0 .

If the result is of type REAL, the result lies in the range ABS(ASIN(*X*)) <= π/2 . If the result is of type COMPLEX, the real part result as a value in radians lies in the range ABS(REAL(ASIN(*X*))) <= π/2 .

The generic name, ASIN, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = asin(.5)
```

2.22 ASIND Intrinsic Function

Description

Arcsine, expressed in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ASIND	---	1	REAL	REAL
	ASIND		Single-prec REAL	Single-prec REAL
	DASIND		Double-prec REAL	Double-prec REAL
	QASIND		Quad-prec REAL	Quad-prec REAL

```
result = ASIND ( X )
```

Argument(s)

X

X shall be of type REAL and shall be in the range -1 <= *X* <= 1 .

Result

The result has the same kind as *X*. Its value is a REAL representation of the arcsine of *X*, expressed in degrees.

Remarks

ASIND, DASIND, and QASIND evaluate the arcsine, expressed in degrees. *X* shall satisfy the inequality ABS(*X*) <= 1.0 . The result has the value 180/π * ASIN(*X*) and lies in the range ABS(ASIND(*X*)) <= 90.0 .

The generic name, ASIND, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = asind(.5)
```

2.23 ASINH Intrinsic Function

Description

Inverse hyperbolic sine.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ASINH	---	1	REAL or COMPLEX	REAL or COMPLEX

```
result = ASINH ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX. If it is type REAL, the domain shall be within the range ABS(*X*) < 1.0 .

Result

The result has the same kind as *X*.

Remarks

ASINH evaluates the inverse hyperbolic sine for type REAL or COMPLEX. If *X* is of type COMPLEX, its imaginary part is treated as a value in radians, and lies in the range 0 <= AIMAG(ASINH(*X*)) <= $\pi/2$.

The type of the result is the same as the type of the argument.

Example

```
r = asinh(0.1)
```

2.24 ASINQ Intrinsic Function

Description

Arcsine, expressed in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ASINQ	---	1	REAL	REAL
	ASINQ		Single-prec REAL	Single-prec REAL
	DASINQ		Double-prec REAL	Double-prec REAL
	QASINQ		Quad-prec REAL	Quad-prec REAL

```
result = ASINQ ( X )
```

Argument(s)

X

X shall be of type REAL and shall be in the range -1 <= *X* <= 1 .

Result

The result has the same kind as *X*. Its value is a REAL representation of the arcsine of *X*, expressed in quadrants.

Remarks

ASINQ, DASINQ, and QASINQ evaluate the arcsine, expressed in quadrants. *X* shall satisfy the inequality ABS(*X*) <= 1.0 . The result has the value 2/π * ASIN(*X*) and lies in the range ABS(ASINQ(*X*)) <= 1.0 .

The generic name, ASINQ, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = asinq(.5)
```

2.25 ASSIGN Statement (deleted feature)

Description

Assigns a statement label to an INTEGER variable.

Syntax

```
ASSIGN label TO scalar-int-variable
```

Where:

label is a statement label.

scalar-int-variable is a scalar INTEGER variable.

Remarks

label shall be the statement label of a branch target statement or FORMAT statement that appears in the same scoping unit as the ASSIGN statement.

scalar-int-variable shall be a named variable of type default INTEGER.

Execution of an ASSIGN statement causes a statement label to be assigned to an INTEGER variable. While defined with a statement label value, the INTEGER variable may be referenced only in the context of an assigned GO TO statement or as a format specifier in an input/output statement.

Example

```
      assign 10 to i
      go to i
10    continue
```

2.26 Assigned GO TO Statement (deleted feature)

Description

The assigned GO TO statement causes a transfer of control to the branch target statement indicated by a variable that was assigned a statement label in an ASSIGN statement.

Syntax

```
GO TO scalar-int-variable [ [ , ] ( label-list ) ]
```

Where:

scalar-int-variable is a scalar INTEGER variable that is named variable of type default INTEGER.

label-list is a comma-separated list of statement labels. Each label in *label-list* shall be the label of a branch target statement in the current scoping unit.

Remarks

At the time of execution of the GO TO statement, *scalar-int-variable* shall be defined with the value of a label of a branch target statement in the same scoping unit. The variable may be defined with a statement label value only by an ASSIGN statement in the same scoping unit as the assigned GO TO statement.

If the parenthesized list of labels is present, the variable shall be one of the labels in the list.

The execution of the assigned GO TO statement causes a transfer of control so that the branch target statement identified by the statement label currently assigned to the *scalar-int-variable* is executed next.

Example

```
      assign 10 to i
      go to i
10    continue
```

2.27 Assignment Statement

Description

Assigns the value of the expression on the right side of the equal sign to the variable on the left side of the equal sign.

Syntax

variable = expr

Where:

variable is a scalar variable, an array, or a variable of derived type. *variable* shall not be a whole assumed-size array.

expr is an expression whose result is conformable with *variable*.

Remarks

An assignment statement is either intrinsic or defined.

A defined assignment statement is an assignment statement that is not an intrinsic assignment statement, and is defined by a subroutine and interface block that specifies ASSIGNMENT(=) (see "1.12.7.3.3 Defined Assignment" and "1.12.7.2 Procedure Interface Block").

An intrinsic assignment statement is an assignment statement where the shapes of *variable* and *expr* conform, if the variable is a nonallocatable array or coarray.

A numeric variable can only be assigned a numeric; a CHARACTER variable can only be assigned a CHARACTER with the same kind; a LOGICAL variable can only be assigned a LOGICAL; and a derived type variable can only be assigned the same derived type. Variables cannot be polymorphic.

If *expr* is array-valued, then *variable* shall be an array. If *expr* is scalar and *variable* is an array, all elements of *variable* are assigned the value of *expr*.

If *variable* is a pointer, it shall be associated with a definable such that the type, type parameters, and shape of the target and *expr* conform. The target is assigned the value of *expr*.

If *variable* and *expr* are numeric types and have different types or different kind type parameter, the value of *expr* is converted to the type and kind type parameter of *variable* according to the following table.

Type of *variable*	Value assigned
INTEGER	INT (*expr* , KIND = KIND (*variable*))
REAL	REAL (*expr* , KIND = KIND (*variable*))
COMPLEX	CMPLX (*expr* , KIND = KIND (*variable*))

The functions INT, REAL, CMPLX, and KIND are the generic intrinsic functions.

If *variable* and *expr* are of LOGICAL type with different kind type parameter, the value of *expr* is converted to the kind type parameter of *variable*.

If *variable* and *expr* are of CHARACTER type with different lengths, *expr* is truncated if longer than *variable*, and padded on the right with blanks if *expr* is shorter than *variable*.

A derived-type intrinsic assignment is performed as if each component of *expr* were assigned to the corresponding component of *variable* using pointer assignment for pointer components, and intrinsic assignment for nonpointer components that are not allocatable. For allocatable components the following sequence of operations is applied:

1. If the component of *variable* is currently allocated, it is deallocated.

2. If the component of *expr* is currently allocated, the corresponding component of *variable* is allocated with the same shape. The value of the component of *expr* is then assigned to the corresponding component of *variable* using intrinsic assignment.

If an allocatable component of *expr* is not currently allocated, the corresponding component of variable has an allocation status of not currently allocated after execution of the assignment.

If a noncoarray *variable* is an unallocated allocatable array, the expressions must have the same rank. If a noncoarray *variable* is allocated, it is deallocated if the expression and *variable* have different shapes or if the *variable* is a character type of unspecified length and the corresponding length type parameter value in the expressions is different.

If the *variable* is an allocatable variable that has not been allocated, or of it becomes an allocate variable that has not been allocated, the *variable* is allocated with its deferred type parameter, format, and each bound the same as the corresponding length type parameter, format, and LBOUND (expression) in the expression. However, if the *variable* is an array and the expression is scalar, the variable bounds remain the same as the bounds immediately preceding.

Example

```
real :: a=1.5, b(10)=1.0
integer :: i=2, j(10)
character (len = 5) :: string5 = "abcde"
character (len = 7) :: string7 = "cdefghi"
type person
     integer :: age
     character (len = 25) :: name
end type person
type (person) :: person1, person2
i = a             ! i is assigned int(a)
i = j             ! error
j = i             ! each element in j assigned
                  ! the value 2
j = b             ! each element in j assigned
                  ! corresponding value in b
                  ! converted to integer
string5 = string7 ! string5 is assigned string7(1:5)
string7 = string5 ! string7 is assigned string5//"  "
person1%age = 5
person1%name = "john"
person2 = person1 ! each component of person2 is
                  ! assigned the value of the
                  ! corresponding component
                  ! of person1
```

2.28 ASSOCIATE Construct

During block execution, the ASSOCIATE construct associates an expression or variable with a named entity. These named entities within a construct are referred to as associate entities, and their names are referred to as associate names.

2.28.1 ASSOCIATE Construct Form

Syntax

```
[ associate-construct-name: ] ASSOCIATE ( association-list )
  block
END ASSOCIATE  [ associate-construct-name ]
```

Where:

association-list is

```
   association => selector
```

association is associate name

selector is

expr or
variable

Remarks

If the selector is not a variable, or if it is a variable with a vector subscript, the associate name cannot appear in a variable definition context.

An associate name cannot be the same as another associate name in the ASSOCIATE statement to which it belongs.

If an ASSOCIATE construct name is specified in an ASSOCIATE statement, the same ASSOCIATE construct name must be specified in the corresponding END ASSOCIATE statement.

If an ASSOCIATE construct name is not specified in an ASSOCIATE statement, an ASSOCIATE construct name cannot be specified in the corresponding END ASSOCIATE statement.

Example

```
real :: a,b
...
associate( c => sin(b) )
print *, a+c
end associate
```

2.28.1.1 ASSOCIATE Construct Execution

When an ASSOCIATE construct is executed, the ASSOCIATE statement within the construct is first executed, followed by the block. During block execution, each of the associate names specifies the objects associated with the corresponding selectors. The associate entities inherit the selector declared types and type parameters. If a selector is polymorphic, the associate entity becomes polymorphic.

A jump to the END ASSOCIATE statement can only be performed from inside that ASSOCIATE construct.

2.28.1.2 Associate Name Attributes

In the SELECT TYPE construct and the ASSOCIATE construct, each associated entity has the same rank as the corresponding selector. The lower bound of each dimension is the result of applying the intrinsic function LBOUND to the rank corresponding to the selector. The upper bound of each dimension is obtained by subtracting 1 from the sum of the lower bound and the extent.

When the selector is a variable with the ASYNCHRONOUS, VOLATILE or TARGET attribute if and only if the associated entity has the same attribute.

When the selector is a variable with the POINTER attribute, the associated entity has the TARGET attribute.

When the selector is CONTIGUOUS, the associating entity is CONTIGUOUS (see "2.82 CONTIGUOUS Statement").

When the associated entity is polymorphic, the dynamic type and type parameter values of that selector are inherited. If the selector has the OPTIONAL attribute, that selector must actually exist.

If the selector cannot appear in a variable definition context, or if it is an array with a vector subscript, the associate name cannot appear in a variable definition context.

2.29 ASSOCIATED Intrinsic Function

Description

Indicate whether a pointer is associated with a target.

Class

Inquiry function.

Syntax

```
ASSOCIATED ( POINTER [ , TARGET ] )
```

Required Argument(s)

POINTER

POINTER shall be a pointer whose pointer association status is not undefined.

Optional Argument(s)

TARGET

TARGET shall be a pointer or target. If it is a pointer, its pointer association status shall not be undefined.

Result

The result is of type default LOGICAL.

Remarks

- If *TARGET* is absent

  The result is true if *POINTER* is currently associated with a target and false if it is not.

- If *TARGET* is specified and it is a procedure

  If *POINTER* is associated with *TARGET*, the result value is true.

- If *TARGET* is specified and it is a procedure pointer

  If the target associated with *POINTER* is associated with the same procedure as *TARGET*, the result is true. If either *POINTER* or *TARGET* is disassociated, the result is false.

- If *TARGET* is specified and it is a scalar

  If the *TARGET* size is not zero and the target associated with *POINTER* specifies the same area as *TARGET*, the result is true. Otherwise it is false.

  If *POINTER* is disassociated, the result value is false.

- If *TARGET* is specified and it is an array

  If *POINTER* and *TARGET* specify the same area for all entities, if the shape is the same and the size is not zero, and if the storage units are the same in the array element sequence, the result value is true. Otherwise it is false.

  If *POINTER* is empty, the result value is false.

- If *TARGET* is specified and it is a scalar pointer

  If the target associated with *POINTER* and the target associated with *TARGET* do not have zero size and specify the same storage unit, the result is true. Otherwise it is false.

  If *POINTER* or *TARGET* is disassociated, the result value is false.

- If *TARGET* is specified and it is array pointer

  If the target associated with *POINTER* and the target associated with *TARGET* have the same shape, do not have zero size, and the storage units are the same in array element order, the result is true. Otherwise it is false.

  If *POINTER* or *TARGET* is disassociated, the result value is false.

Example

```
real, pointer :: a, b, e
real, target :: c, f
logical :: l
a => c
b => c
e => f
l = associated (a)     ! l is assigned the value true
l = associated (a, c)  ! l is assigned the value true
l = associated (a, b)  ! l is assigned the value true
l = associated (a, f)  ! l is assigned the value false
l = associated (a, e)  ! l is assigned the value false
```

2.30 ASYNCHRONOUS Statement

Description

This statement specifies the ASYNCHRONOUS attribute for objects in the list.

Syntax

```
ASYNCHRONOUS [ :: ] object-name-list
```

Remarks

The ASYNCHRONOUS attribute specifies whether or not variables are handled by asynchronous input/output.

If both the following conditions are met, variables must have the ASYNCHRONOUS attribute within the scope:

- The variables appear in execution statements or specification expressions within the scope, and
- The variables are related to asynchronous input/output and any statements within the scope are currently being executed.

If an object has the ASYNCHRONOUS attribute, all the sub-objects have the ASYNCHRONOUS attribute.

If a variable is specified in any of the items below in an ASYNCHRONOUS input/output statement, the ASYNCHRONOUS attribute is implicitly given to that object within the scope of that data transfer statement. This attribute can be defined in an explicit declaration.

- Input/output list
- Namelist group object
- SIZE= specifier

Even if a module entity with use association does not have the ASYNCHRONOUS attribute, it can have the ASYNCHRONOUS attribute within the local scope.

Example

```
asynchronous:: a
```

2.31 ATAN Intrinsic Function

Description

Arctangent, expressed in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ATAN	---	1	REAL or COMPLEX	REAL or COMPLEX
	ATAN		Single-prec REAL	Single-prec REAL
	DATAN		Double-prec REAL	Double-prec REAL
	QATAN		Quad-prec REAL	Quad-prec REAL
	---	2	REAL , REAL	REAL

```
result = ATAN ( X ) or
result = ATAN ( Y, X )
```

Argument(s)

Y

Y shall be of type REAL.

X

If *Y* is specified, *X* shall be of type REAL of the same kind as *Y*. If *Y* is not specified, *X* shall be of type REAL or COMPLEX.

Result

The result is of the same kind as *X*.

Remarks

ATAN evaluates the arctangent, expressed in radians for type REAL or COMPLEX. DATAN and QATAN evaluate the arctangent, expressed in radians for type REAL.

If *Y* has the value zero, *X* shall not have the value zero.

If *Y* is specified, the result is same as ATAN2 (*Y*, *X*). If *Y* is not specified, the result lies in the range ABS(ATAN(*X*)) <= π /2 .

The generic name, ATAN, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
a = atan(.5)
```

2.32 ATAN2 Intrinsic Function

Description

Arctangent of *y/x* (principal value of the argument of the COMPLEX number (*x*,*y*)), expressed in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ATAN2	---	2	REAL , REAL	REAL
	ATAN2		Single-prec REAL , Single-prec REAL	Single-prec REAL
	DATAN2		Double-prec REAL , Double-prec REAL	Double-prec REAL
	QATAN2		Quad-prec REAL , Quad-prec REAL	Quad-prec REAL

```
result = ATAN2 ( Y , X )
```

Argument(s)

Y

Y shall be of type REAL.

X

X shall be of the same kind as *Y*. If *Y* has the value zero, *X* shall not have the value zero.

Result

The result is of the same kind as *X*. Its value is a REAL representation, expressed in radians, of the argument of the COMPLEX number (*X*, *Y*).

Remarks

ATAN2, DATAN2, and QATAN2 evaluate the arctangent, expressed in radians. If *Y* and *X* are not both zero, the result has a value ATAN(*Y* / *X*), otherwise it is undefined. The result lies in the range ABS(ATAN2(*Y*,*X*)) <= π .

The generic name, ATAN2, may be used with any REAL arguments.

The type of the result of each function is the same as the type of the arguments.

Example

```
x = atan2 (1., 1.)
```

2.33 ATAN2D Intrinsic Function

Description

Arctangent of *y*/*x* (principal value of the argument of the COMPLEX number (*x*,*y*)), expressed in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ATAN2D	---	2	REAL , REAL	REAL
	ATAN2D		Single-prec REAL , Single-prec REAL	Single-prec REAL
	DATAN2D		Double-prec REAL , Double-prec REAL	Double-prec REAL
	QATAN2D		Quad-prec REAL , Quad-prec REAL	Quad-prec REAL

result = ATAN2D (*Y* , *X*)

Argument(s)

Y

Y shall be of type REAL.

X

X shall be of the same kind as *Y*. If *Y* has the value zero, *X* shall not have the value zero.

Result

The result is of the same kind as *X*. Its value is a REAL representation, expressed in degrees, of the argument of the COMPLEX number (*X* , *Y*).

Remarks

ATAN2D, DATAN2D, and QATAN2D evaluate the arctangent, expressed in degrees. If *Y* and *X* are not both zero, the result has a value ATAND(*Y* / *X*), otherwise it is undefined. The result lies in the range ABS(ATAN2D(*Y*,*X*)) <= 180.0 .

The generic name, ATAN2D, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
x = atan2d (1., 1.)
```

2.34 ATAN2Q Intrinsic Function

Description

Arctangent of *y* / *x* (principal value of the argument of the COMPLEX number (*x*,*y*)), expressed in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ATAN2Q	---	2	REAL , REAL	REAL
	ATAN2Q		Single-prec REAL , Single-prec REAL	Single-prec REAL
	DATAN2Q		Double-prec REAL , Double-prec REAL	Double-prec REAL
	QATAN2Q		Quad-prec REAL , Quad-prec REAL	Quad-prec REAL

```
result = ATAN2Q ( Y , X )
```

Argument(s)

Y

Y shall be of type REAL.

X

X shall be of the same kind as *Y*. If *Y* has the value zero, *X* shall not have the value zero.

Result

The result is of the same kind as *X*. Its value is a REAL representation, expressed in quadrants, of the argument of the COMPLEX number (*X* , *Y*).

Remarks

ATAN2Q, DATAN2Q, and QATAN2Q evaluate the arctangent, expressed in degrees. If *Y* and *X* are not both zero, the result has a value ATANQ(*Y* / *X*), otherwise it is undefined. The result lies in the range ABS(ATAN2Q(*Y*,*X*)) <= 1.0 .

The generic name, ATAN2Q, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
x = atan2q (1., 1.)
```

2.35 ATAND Intrinsic Function

Description

Arctangent, expressed in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ATAND	---	1	REAL	REAL
	ATAND		Single-prec REAL	Single-prec REAL
	DATAND		Double-prec REAL	Double-prec REAL
	QATAND		Quad-prec REAL	Quad-prec REAL

result = ATAND (*X*)

Argument(s)

X

X shall be of type REAL.

Result

The result is a REAL representation of the arctangent of *X*, expressed in degrees, that lies within the range -90.0 <= *X* <= 90.0 .

Remarks

ATAND, DATAND, and QATAND evaluate the arctangent, expressed in degrees. The result lies in the range ABS(ATAND(*X*)) <= 90.0 .

The generic name, ATAND, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
a = atand(.5)
```

2.36 ATANH Intrinsic Function

Description

Inverse hyperbolic tangent.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ATANH	---	1	REAL or COMPLEX	REAL or COMPLEX

result = ATANH (*X*)

Argument(s)

X

X shall be of type REAL or COMPLEX. If it is of type REAL, the domain shall be within the range ABS(*X*) < 1.0 .

Result

The result has the same kind as *X*.

Remarks

ATANH evaluates the inverse hyperbolic tangent for type REAL or COMPLEX.

If *X* is of type COMPLEX, its imaginary part is treated as a value in radians, and lies in the range ABS(AIMAG(ATANH(*X*))) <= $\pi/2$.

The type of the result is the same as the type of the argument.

Example

```
r = atanh(0.76)
```

2.37 ATANQ Intrinsic Function

Description

Arctangent, expressed in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ATANQ	---	1	REAL	REAL
	ATANQ		Single-prec REAL	Single-prec REAL
	DATANQ		Double-prec REAL	Double-prec REAL
	QATANQ		Quad-prec REAL	Quad-prec REAL

```
result = ATANQ ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is a REAL representation of the arctangent of *X*, expressed in quadrants, that lies within the range -1.0 <= *X* <= 1.0 .

Remarks

ATANQ, DATANQ, and QATANQ evaluate the arctangent, expressed in quadrants. The result lies in the range ABS(ATANQ(*X*)) <= 1.0 .

The generic name, ATANQ, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
a = atanq(.5)
```

2.38 ATOMIC_DEFINE Intrinsic Subroutine

Description

Defines a variable atomically.

Class

Atomic subroutine.

Syntax

```
CALL ATOMIC_DEFINE ( ATOM, VALUE [, STAT ] )
```

Required Argument(s)

ATOM

ATOM must be a scalar coarray or a coindexed object. It must be of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND. It is an INTENT(OUT) argument. If *VALUE* is of type integer with kind ATOMIC_INT_KIND or of type logical, *ATOM* is defined with the value of *VALUE*. Otherwise, *ATOM* is defined with the value of INT(*VALUE*, ATOMIC_INT_KIND).

VALUE

VALUE must be an integer scalar if *ATOM* is integer, and must be logical scalar if *ATOM* is logical. It is an INTENT(IN) argument.

Optional Argument(s)

STAT

STAT must be a noncoindexed integer scalar variable. It is an INTENT(OUT) argument. If the execution terminates successfully, 0 is assigned to the variable. If error occurs during the execution, the diagnostic message number is assigned to the variable.

Example

```
use iso_fortran_env, only: atomic_int_kind
integer(atomic_int_kind), save:: k[*]
call atomic_define(k[1],1) ! 1 is assigned to variable k in image1
```

2.39 ATOMIC_REF Intrinsic Subroutine

Description

References a variable atomically.

Class

Atomic subroutine.

Syntax

```
CALL ATOMIC_REF ( VALUE, ATOM [, STAT ] )
```

Required Argument(s)

VALUE

VALUE must be an integer scalar if *ATOM* is integer, and must be a logical scalar if *ATOM* is logical. It is an INTENT(OUT) argument. If *VALUE* is of type integer with kind ATOMIC_INT_KIND or of type logical, *VALUE* is defined with the value of *ATOM*. Otherwise, *VALUE* is defined with the value of INT(*ATOM*, KIND(*VALUE*)).

ATOM

ATOM must be a scalar coarray or a coindexed object. It must be of type integer with kind ATOMIC_INT_KIND or of type logical with kind ATOMIC_LOGICAL_KIND. It is an INTENT(IN) argument.

Optional Argument(s)

STAT

STAT must be a noncoindexed scalar integer variable. It is an INTENT(OUT) argument. If the execution terminates successfully, 0 is assigned to the variable. If error occurs during the execution, the diagnostic message number is assigned to the variable.

Example

```
use iso_fortran_env, only: atomic_int_kind
integer(atomic_int_kind) :: k[*], v
save:: k
k=0
sync all
call atomic_ref(v,k[1]) ! 0 is assigned to variable v
```

2.40 AUTOMATIC Statement

Description

The AUTOMATIC statement declares specified variables to be on the stack. It also makes the specified variables become undefined outside the subprogram when the subprogram exits through a RETURN or END statement.

Syntax

```
AUTOMATIC [ [ :: ] object-name-list ]
```

Where:

object-name-list is a comma-separated list of an object name.

object-name shall not be a dummy argument, a procedure, a function result, an automatic object, an equivalence object, a coarray, or a common block object.

Remarks

If *object-name-list* is not present in the AUTOMATIC statement, all the allowable local variables without SAVE attribute are implicitly automatic.

An AUTOMATIC statement may not be specified in a module specification part and block data program unit. An AUTOMATIC statement in the main program has no effect.

Example

```
subroutine sub
  automatic :: i,j  ! i and j are on the stack
```

2.41 BACKSPACE Statement

Description

The BACKSPACE statement positions the file before the current record if there is a current record, otherwise before the preceding record.

Execution of the BACKSPACE statement performs a wait operation on all unfinished asynchronous data transfer operations at the specified device.

Syntax

```
BACKSPACE external-file-unit          or
BACKSPACE ( position-spec-list )
```

Where:

external-file-unit is a scalar INTEGER expression corresponding to the input/output unit number of an external file.

position-spec-list is a comma-separated list of

```
[ UNIT = ] external-file-unit          or
IOMSG = iomsg                          or
IOSTAT = io-stat                       or
ERR = err-label
```

position-spec-list shall contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

If the optional characters UNIT= are omitted from the *external-file-unit* specifier, the *external-file-unit* specifier shall be the first item in the *position-spec-list*.

io-stat is a variable of type INTEGER that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, and zero otherwise.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the BACKSPACE statement. If an error condition occurs during execution of the BACKSPACE statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

Remarks

If there is a current record, the file is positioned before the current record. If there is no current record, the file is positioned before the preceding record. If there is no current record and no preceding record, the file position is not changed.

If the preceding record is an endfile record, the file is positioned before the endfile record.

If the BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before the record that precedes the endfile record.

To backspace a file that is connected but does not exist is prohibited.

To backspace over records written using list-directed or namelist formatting is prohibited.

iomsg must be a scalar default CHARACTER variable. If the error condition occurs during execution of the BACKSPACE statement, an explanatory message is assigned to *iomsg*.

The *iomsg* value does not change if no error conditions occur.

Files connected as direct access or as unformatted stream access are not referenced by the BACKSPACE statement.

Example

```
backspace 10  ! file connected to unit 10 backspaced
backspace (10, err = 100)
              ! file connected to unit 10 backspaced
              ! on error go to label 100
```

2.42 BGE Intrinsic Function

Description

Bit-wise comparison.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
BGE	---	2	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant	Default LOGICAL

result = BGE (*I* , *J*)

Argument(s)

I

I shall be of type INTEGER, or a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER, or a binary, octal or hexadecimal constant.

Result

The result is of type default LOGICAL.

Remarks

The result is true if the sequence of bits represented by *I* is greater than or equal to the sequence of bits represented by *J*, otherwise the result is false.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. If the binary, octal or hexadecimal constants are specified for both, the constants are converted to default INTEGER.

The result is of type default LOGICAL.

Example

```
logical :: l
l = bge(-1,1)    ! l is assigned the value TRUE
```

2.43 BGT Intrinsic Function

Description

Bit-wise comparison.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
BGT	---	2	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant	Default LOGICAL

```
result = BGT ( I , J )
```

Argument(s)

I

I shall be of type INTEGER, or a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER, or a binary, octal or hexadecimal constant.

Result

The result is of type default LOGICAL.

Remarks

The result is true if the sequence of bits represented by *I* is greater than the sequence of bits represented by *J*, otherwise the result is false.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. If the binary, octal or hexadecimal constants are specified for both, the constants are converted to default INTEGER.

The result is of type default LOGICAL.

Example

```
logical :: l
l = bgt(-1,1)    ! l is assigned the value TRUE
```

2.44 BIC Service Subroutine

Description

BIC sets bit *pos* of *i* to 0.

Syntax

```
CALL  BIC ( pos , i )
```

Argument(s)

pos

Default INTEGER scalar. Bit number to set.

It shall be nonnegative and less than or equal to BIT_SIZE(*i*).

i

Default INTEGER. Variable whose bit is to be set.

Example

```
use service_routines,only:bic
integer :: i
i = -1
call bic(31,i)
write(6,fmt="(1x,z8.8)") i      ! Output 7FFFFFFF
end
```

2.45 BIND Statement

Syntax

language-binding-spec [::] *bind-entity-list*

language-binding-spec is

BIND (C [, NAME = *scalar-char-initialization-expr*])

bind-entity-list is

entity-name or
/ *common-block-name* /

A *scalar-char-initialization-expr* must be the default CHARACTER type kind.

When a value that ignores contiguous spaces at the start and end of a *scalar-char-initialization-expr* has a length of 1 or greater, it is not a valid identifier on the companion processor side.

Items that have the BIND attribute automatically have the SAVE attribute. However, the SAVE statement can be specified to explicitly specify the SAVE attribute.

Even if the BIND attribute is specified for a data entity, it does not necessarily have any effect at the companion processor.

If an entity name is included in a bind entity in a BIND statement, the BIND statement must appear in the module declaration part and the entity must be an interoperable variable. The variable must not be a variable with POINTER attribute, ALLOCATBLE attribute, or a coarray.

If the language binding specifier is a NAME= specifier, there must be one entity in the bind entity list.

If the bind entity is a common block, every variable in the common block must be interoperable.

This statement specifies the BIND attribute for variables in a list or for common blocks.

2.46 BIS Service Subroutine

Description

BIS sets bit *pos* of *i* to 1.

Syntax

CALL BIS (*pos* , *i*)

Argument(s)

pos

Default INTEGER scalar. Bit number to set.

It shall be nonnegative and less than or equal to BIT_SIZE(*i*).

i

Default INTEGER. Variable whose bit is to be set.

Example

```
use service_routines,only:bis
integer :: i
i = 1
call bis(31,i)
write (6,fmt="(1x,z8.8)") i   ! output 80000001
end
```

2.47 BIT Service Function

Description

Performs a bit-level test for integers.

Syntax

```
ly = BIT ( pos , i )
```

Argument(s)

pos

Default INTEGER scalar. Bit number to test.

i

Default INTEGER scalar. Variable tested.

Result

Default LOGICAL scalar. .TRUE. if bit *pos* of *i* is 1; otherwise, .FALSE.. .FALSE. if *pos* is negative or less than BIT_SIZE(*i*).

Example

```
use service_routines,only:bit
logical :: l(2)
i = 10
l = (/bit(0,i),bit(1,i)/)     ! l is assigned the value (/.false.,.true./)
end
```

2.48 BIT_SIZE Intrinsic Function

Description

Size, in bits, of a data object of type INTEGER.

Class

Inquiry function.

Syntax

```
result = BIT_SIZE ( I )
```

Argument(s)

I

I shall be of type INTEGER. It can be scalar or array-valued.

Result

The result has the same kind as *I*. Its value is equal to the number of bits in *I*.

The result value is as follows:

Type of *I*	The result value
One-byte INTEGER	8_1
Two-byte INTEGER	16_2
Four-byte INTEGER	32_4
Eight-byte INTEGER	64_8

Example

```
integer :: i, m, n
integer, dimension (2) :: j
m = bit_size (i) ! m is assigned the value 32_4
n = bit_size (j) ! n is assigned the value 32_4
```

2.49 BLE Intrinsic Function

Description

Bit-wise comparison.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
BLE	---	2	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant	Default LOGICAL

```
result = BLE ( I , J )
```

Argument(s)

I

I shall be of type INTEGER, or a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER, or a binary, octal or hexadecimal constant.

Result

The result is of type default LOGICAL.

Remarks

The result is true if the sequence of bits represented by *I* is less than or equal to the sequence of bits represented by *J*, otherwise the result is false.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. If the binary, octal or hexadecimal constants are specified for both, the constants are converted to default INTEGER.

The result is of type default LOGICAL.

Example

```
logical :: l
l = ble(-1,1)    ! l is assigned the value FALSE
```

2.50 BLOCK Construct

Description

The BLOCK construct is an executable construct that may include declarations.

Syntax

```
[ block-construct-name : ] BLOCK
   [ specification-part ]
   block
 END BLOCK [ block-construct-name ]
```

Where:

block-construct-name is an optional name given to the BLOCK construct. If the BLOCK statement is identified by a *block-construct-name* , the corresponding END BLOCK statement must specify the same *block-construct-name*. If the BLOCK statement is not identified by a *block-construct-name*, the corresponding END BLOCK statement must not specify a *block-construct-name*.

specification-part is a sequence of zero of more specification statements.

block is a sequence of zero or more statements or executable constructs.

Remarks

Followings must not appear within BLOCK constructs:

- parameterized derived type definition, object declaration, or component declaration (see "1.5.11.2 Derived Type Parameter"), or
- declaration of coarray (see "1.17 Coarray"), or
- ASSIGN statement (see "2.25 ASSIGN Statement (deleted feature)"), or
- assigned GO TO statement (see "2.26 Assigned GO TO Statement (deleted feature)"), or
- COMMON statement (see "2.75 COMMON Statement"), or
- EQUIVALENCE statement (see "2.153 EQUIVALENCE Statement"), or
- IMPLICIT statement (see "2.267 IMPLICIT Statement"), or
- INTENT statement (see "2.275 INTENT Statement"), or
- NAMELIST statement (see "2.344 NAMELIST Statement"), or
- OPTIONAL statement (see "2.355 OPTIONAL Statement"), or
- statement function statement (see "2.441 Statement Function Statement (obsolescent feature)"), or
- USE statement (see "2.479 USE Statement"), or
- VALUE statement (see "2.481 VALUE Statement").

SAVE statement in BLOCK construct must have entity list that does not specify a common block name.

Specification statement in BLOCK construct declares entity whose scope is the BLOCK construct, except ASYNCHRONOUS (see "2.30 ASYNCHRONOUS Statement") and VOLATILE (see "2.483 VOLATILE Statement") statements.

A branch out of a BLOCK construct must not have a branch target in the BLOCK construct.

Example

```
k = 5
block
  integer k ! k has a scope of BLOCK construct
  k = 10
  print *,k ! 10 is printed
end block
print *,k   ! 5 is printed
```

2.51 BLOCK DATA Statement

Description

The BLOCK DATA statement begins a block data program unit. See "1.11.3 Block Data Program Units" for block data program unit.

Syntax

```
BLOCK DATA [ block-data-name ]
```

Where:

block-data-name is an optional name given to the block data program unit.

Remarks

block-data-name is global entities of a program, it shall not be used to identify other program units, external procedures, and common blocks in the same program. Also it shall not be used to identify a local entity in the block data program unit.

There shall not be more than one unnamed block data program unit in a program.

Example

```
block data
  common /com/ a, b, c
  integer :: a=1, b=2, c=3
end block data
```

2.52 BLT Intrinsic Function

Description

Bit-wise comparison.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
BLT	---	2	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant	Default LOGICAL

```
result = BLT ( I , J )
```

Argument(s)

I

I shall be of type INTEGER, or a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER, or a binary, octal or hexadecimal constant.

Result

The result is of type default LOGICAL.

Remarks

The result is true if the sequence of bits represented by *I* is less than the sequence of bits represented by *J*, otherwise the result is false.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. If the binary, octal or hexadecimal constants are specified for both, the constants are converted to default INTEGER.

The result is of type default LOGICAL.

Example

```
logical :: l
l = blt(-1,1)    ! l is assigned the value FALSE
```

2.53 BTEST Intrinsic Function

Description

Test a bit of an INTEGER data object.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
BTEST	---	2	One-byte INTEGER	Default LOGICAL
	BITEST		Two-byte INTEGER	Default LOGICAL
	BTEST		Four-byte INTEGER	Default LOGICAL
	BJTEST		Four-byte INTEGER	Default LOGICAL
	---		Eight-byte INTEGER	Default LOGICAL

result = BTEST (*I* , *POS*)

Argument(s)

I

I shall be of type INTEGER.

POS

POS shall be of type INTEGER. It shall be non-negative and less than BIT_SIZE (*I*). Bits are numbered from least significant to most significant, beginning with 0.

Result

The result is of type default LOGICAL. It has the value true if bit *POS* has the value 1 and false if bit *POS* has the value zero.

Example

```
logical l
l = btest (1, 0)     ! l is assigned the value true
l = btest (4, 1)     ! l is assigned the value false
l = btest (32, 5)    ! l is assigned the value true
```

2.54 BYTE Type Declaration Statement

Description

The BYTE type declaration statement declares entities of type one-byte INTEGER. See "2.469 Type Declaration Statement" for type declaration statement.

Syntax

```
BYTE [ [ , attr-spec ] ... :: ] entity-decl-list
```

2.55 CALL Statement

Description

The CALL statement invokes a subroutine and passes to it a list of arguments.

Syntax

```
CALL procedure-designator [ ( [ actual-arg-spec-list ] ) ]
```

procedure-designator is

```
procedure-name              or
proc-component-ref          or
data-ref % binding-name
```

Where:

procedure-name must be the name of a procedure or a procedure pointer.

binding-name must be the binding name of the *data-ref* declared type.

If *data-ref* is an array, the type bound procedure being referenced must have the PASS attribute.

actual-arg-spec-list is a comma-separated list of

```
[ keyword = ] actual-arg
```

keyword is the name of a dummy argument to *subroutine-name*. *keyword* shall not be present if the *subroutine-name* has an implicit interface. The *keyword* = may be omitted from an *actual-arg-spec* only if the *keyword* = has been omitted from each preceding *actual-arg-spec* in the argument list. Each *keyword* shall be the name of a dummy argument in the explicit interface of the *subroutine-name*.

actual-arg is an actual argument that is

```
expr                or
variable            or
procedure-name      or
*label              or
%VAL ( expr )
```

expr is an expression.

variable is a variable.

procedure-name is a name of a procedure. *procedure-name* shall not be a non-intrinsic elemental procedure name. *procedure-name* shall not be the name of an internal procedure or of a statement function and shall not be the generic name of procedure unless it is also a specific name.

label is a statement label. *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the CALL statement.

%VAL specifier passes argument by value.

Remarks

The type, type parameters, and shape of the actual argument shall be consistent with the characteristics of the dummy arguments. See "1.12.6 Procedure Reference" for argument association.

Example

```
x = 3.0
call alpha (x, y)
end program
subroutine alpha (a, b)
  implicit none
  real, intent(in) :: a
  real, intent(out) :: b
  ...
end subroutine alpha
```

2.56 CASE Construct

Description

The CASE construct selects for execution at most one of its constituent blocks.

Syntax

```
[ case-construct-name : ] SELECT CASE ( case-expr )
      [ CASE case-selector [ case-construct-name ]
            block ] ...
END SELECT [ case-construct-name ]
```

Where:

case-construct-name is an optional name for the CASE construct. If the SELECT CASE statement is identified by a *case-construct-name*, the corresponding END SELECT statement shall specify the same *case-construct-name*. If the SELECT CASE statement is not identified by a *case-construct-name*, the corresponding END SELECT statement shall not specify a *case-construct-name*. If a CASE statement is identified by a *case-construct-name*, the corresponding SELECT CASE statement shall specify the same *case-construct-name*.

case-expr is a scalar expression of type INTEGER, LOGICAL, or CHARACTER.

case-selector is

```
( case-value-range-list )        or
DEFAULT
```

At most one DEFAULT selector is allowed in a given case construct.

case-value-range-list is comma-separated list of

```
case-value                 or
case-value :               or
: case-value               or
case-value : case-value
```

case-value is a scalar initialization expression of type INTEGER, LOGICAL, or CHARACTER.

For a given case construct, each *case-value* shall be of the same type as *case-expr* in the SELECT CASE statement. For character type, length differences are allowed, but the kind type parameters shall be the same. If *case-expr* is of type LOGICAL, a *case-value-range* using a colon shall not be used.

The ranges of *case-values* in a case construct shall not overlap; that is, there shall be no possible value of the *case-expr* that matches more than one *case-value-range*.

block is a sequence of zero or more executable statements and constructs. *block* need not contain any executable statements and constructs. Execution of such a *block* has no effect.

Remarks

Execution of a SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the case index. If the case index is in the range specified with a CASE statement's *case-selector*, the *block* following the CASE statement, if any, is executed. For a case index with a value of *c*, a matches are determined as follows:

- If the *case-value-range* contains a single value *v* without a colon, a match occurs for data type LOGICAL if the expression *c* .EQV. *v* is true, and a match occurs for data type INTEGER or CHARACTER if the expression *c* == *v* is true.
- If the *case-value-range* is of the form *low* : *high*, a match occurs if the expression *low* <= *c* .AND. *c* <= *high* is true.
- If the *case-value-range* is of the form *low* : a match occurs if the expression *low* <= *c* is true.
- If the *case-value-range* is of the form : *high*, a match occurs if the expression *c* <= *high* is true.

If no other selector matches and a DEFAULT selector is present, it matches the case index.

If no other selector matches and the DEFAULT selector is absent, there is no match. Execution of the CASE construct completes any *block*s is not executed.

Example

```
select case (i)
case (:-2)
  print *, "i is less than or equal to -2"
case (0)
  print *, "i is equal to 0"
case (1:97)
  print *, "i is in the range 1 to 97, inclusive"
case default
  print *, "i is either -1 or greater than 97"
end select
```

2.57 CASE Statement

Description

The CASE statement shall be specified in a CASE construct, and specifies the condition to execute the following block. See "2.56 CASE Construct" for CASE construct.

Syntax

```
CASE case-selector [ case-construct-name ]
```

Where:

case-selector is

```
( case-value-range-list )     or
DEFAULT
```

At most one DEFAULT selector is allowed in a given case construct.

case-value-range-list is comma-separated list of

```
case-value                    or
case-value :                  or
: case-value                  or
case-value : case-value
```

case-value is a scalar initialization expression of type INTEGER, LOGICAL, or CHARACTER.

construct-name is an optional name assigned to the construct.

2.58 CBRT Intrinsic Function

Description

Cube Root.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
CBRT	---	1	REAL	REAL
	CBRT		Single-prec REAL	Single-prec REAL
	DCBRT		Double-prec REAL	Double-prec REAL
	QCBRT		Quad-prec REAL	Quad-prec REAL

```
result = CBRT ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same kind and type as *X*. The result value is a REAL representation of the cube root of *X*.

Remarks

CBRT, DCBRT, and QCBRT evaluate the cube root of REAL data.

The generic name, CBRT, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
real :: r
r = cbrt(27.0) ! r is assigned the value 3.0
```

2.59 CEILING Intrinsic Function

Description

Smallest INTEGER greater than or equal to a number.

Class

Elemental function.

Syntax

```
result = CEILING ( A [ , KIND ] )
```

Required Argument(s)

A

A shall be of type REAL.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is an INTEGER whose value is the smallest integer greater than or equal to *A*. If *KIND* is present, the kind is that specified by *KIND*. If *KIND* is absent, the kind is that of the default INTEGER type.

Example

```
i = ceiling (-4.7)  ! i is assigned the value -4
i = ceiling (4.7)   ! i is assigned the value 5
```

2.60 CHANGEENTRY Statement

The CHANGEENTRY statement changes rules for processing Fortran procedure names.

Syntax

```
CHANGEENTRY [ :: ] external-proc-name-list
```

Where:

external-proc-name-list is comma-separated list of external procedure name.

Remarks

If *external-proc-name* is the name of the procedure that contains the CHANGEENTRY statement, the rules for processing this defined procedure name is changed. Otherwise, *external-proc-name* has an EXTERNAL attribute implicitly.

Cannot be specified with BIND attribute.

See "Fortran User's Guide" for processing Fortran procedure names.

Example

```
changeentry :: csub
call csub()
end
```

2.61 CHAR Intrinsic Function

Description

Given character in the collating sequence of a given character set.

Class

Elemental function.

Syntax

```
CHAR ( I [ , KIND ] )
```

Required Argument(s)

I

I shall be of type INTEGER. It shall be positive and not greater than the number of characters in the collating sequence of the character set specified by *KIND*.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is a CHARACTER of length one corresponding to the *I*th character of the given character set.

If the kind type parameter *KIND* is specified, the function result type is the CHARACTER type of length 1 and the size of *KIND*. If *KIND* is omitted, the function result type is the default CHARACTER type of length 1.

Example

```
c = char(65)  ! char is assigned the value 'A'
              ! with ASCII the default character type
```

2.62 CHARACTER Type Declaration Statement

Description

The CHARACTER type declaration statement declares entities of type CHARACTER. See "2.469 Type Declaration Statement" for type declaration statement.

Syntax

```
CHARACTER [ char-selector ] [ [ , attr-spec ] ... :: ] entity-decl-list
```

2.63 CHDIR Service Function

Description

Changes the default directory.

Syntax

```
iy = CHDIR ( dirname )
```

Argument(s)

dirname

Default CHARACTER scalar. Name of a directory to become the current directory.

Result

Default INTEGER scalar. Zero if the directory was changed successfully; otherwise, a system error code.

Example

```
use service_routines,only:chdir
integer :: iy
iy = chdir("/tmp")
end
```

2.64 CHMOD Service Function

Description

Changes the attributes of a file.

Syntax

```
iy  = CHMOD ( fname , mode )
```

Argument(s)

fname

Default CHARACTER scalar. Name of the file whose access mode is to be changed.

mode

Default CHARACTER scalar. File permission.

Result

Default INTEGER scalar. Zero if the mode was changed successfully; otherwise, a system error code.

Example

```
use service_routines,only:chmod
character(len=8) :: fname="test.f90"
write(6,*) chmod(fname, "600")
end
```

2.65 CLASS Type Declaration Statement

Description

The CLASS type declaration statement declares a polymorphic object. See "2.469 Type Declaration Statement".

Syntax

```
CLASS ( type-name ) [  [ , attr-spec ]... :: ] entity-decl-list        or
CLASS ( * ) [  [ , attr-spec ]... :: ] entity-decl-list
```

2.66 CLASS DEFAULT Statement

Description

The CLASS DEFAULT statement shall be specified in a SELECT TYPE construct. See "2.413 SELECT TYPE Construct" for SELECT TYPE construct.

Syntax

```
CLASS DEFAULT [ select-construct-name ]
```

Where:

select-construct-name is an optional name assigned to the SELECT TYPE construct.

2.67 CLASS IS Statement

Description

The CLASS IS statement shall be specified in a SELECT TYPE construct. See "2.413 SELECT TYPE Construct" for SELECT TYPE construct.

Syntax

```
CLASS IS (derived-type-spec ) [ select-construct-name ]
```

Where:

derived-type-spec is a derived type specifier. See "1.5.11.8 Derived Type Specifier" for derived type specifier.

derived-type-spec must specify that each length type parameter is inherited.

derived-type-spec cannot specify a sequential derived type or a type having the BIND attribute.

derived-type-spec shall be name of the derived type being defined.

select-construct-name is an optional name assigned to the SELECT TYPE construct.

2.68 CLOCK Service Subroutine

Description

CLOCK returns CPU time for the calling process since the start of execution.

Syntax

```
CALL  CLOCK ( g , i1 , i2 )
```

Argument(s)

g

Four-byte INTEGER, eight-byte INTEGER, single precision REAL, double precision REAL, or quadruple precision REAL. It is a scalar. Variable whose CPU time is returned in *i1.*

i1

Default INTEGER scalar. Default value is 0. Unit to return.

```
=0        :        In seconds.
=1        :        In milliseconds.
=2        :        In microseconds.
```

i2

Default INTEGER scalar. Default value is 0. Type to return.

```
=0        :        INTEGER(4)
=1        :        REAL(4)
=2        :        REAL(8)
=3        :        REAL(16)
=4        :        INTEGER(8)
```

Example

```
use service_routines,only:clock
real :: g,g1
call clock(g,2,1)
do i=1,30
  write(10,*)i,i*i
end do
call clock(g1,2,1)
print *,'CPU TIME (in micro second) = ', g1-g
end
```

2.69 CLOCKM Service Subroutine

Description

CLOCKM returns CPU time, in seconds, for the calling process since the start of execution.

Syntax

```
CALL  CLOCKM ( i )
```

Argument(s)

i

INTEGER(4). It is a scalar. Variable whose CPU time is returned in milliseconds.

Example

```
use service_routines,only:clockm
integer :: t,t1
call clockm(t)
do i=1,30
  write(10,*) i,i*i
end do
call clockm(t1)
print *,'CPU TIME (in milli second) = ', t1-t
end
```

2.70 CLOCKV Service Subroutine

Description

CLOCKV returns CPU time for the calling process since the start of execution.

Syntax

```
CALL  CLOCKV ( g1 , g2 , i1 , i2 )
```

Argument(s)

g1

Four-byte INTEGER, eight-byte INTEGER, single precision REAL, double precision REAL, or quadruple precision REAL. It is a scalar. Variable 0 is returned always.

g2

Four-byte INTEGER, eight-byte INTEGER, single precision REAL, double precision REAL, or quadruple precision REAL. It is a scalar. Variable whose CPU time is returned in *i1*.

i1

Default INTEGER scalar. Default value is 0. Unit to return.

```
=0        :         In seconds.
=1        :         In milliseconds.
=2        :         In microseconds.
```

i2

Default INTEGER scalar. Default value is 0. Type to return.

```
=0        :        INTEGER(4)
=1        :        REAL(4)
=2        :        REAL(8)
=3        :        REAL(16)
=4        :        INTEGER(8)
```

Example

```
use service_routines,only:clockv
integer :: i
real :: g1,g2,g3,g4
call clockv(g1,g2,2,1)
do i=1,30
  write(10,*)i,i*i
end do
call clockv(g3,g4,2,1)
print *,'CPU TIME (in micro second) = ', g4-g2
end
```

2.71 CLOSE Statement

Description

The CLOSE statement terminates the connection of a specified unit to an external file.

Syntax

```
CLOSE ( close-spec-list )
```

Where:

close-spec-list is a comma-separated list of

```
[ UNIT = ] external-file-unit        or
IOMSG = iomsg                        or
IOSTAT = io-stat                     or
ERR = err-label                      or
STATUS = status
```

close-spec-list shall contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

external-file-unit is the input/output unit number of an external file, it shall be an expression of type INTEGER. If UNIT= is omitted, *external-file-unit* shall be the first specifier in *close-spec-list*.

iomsg must be a scalar default CHARACTER variable. If the error condition occurs during input/output statement execution, an explanatory message is assigned to *iomsg*.

If an error condition does not occur, the *iomsg* value does not change.

io-stat is a scalar INTEGER variable. If present, it is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs in executing the CLOSE statement and the program is not terminated; if no error condition occurs it is assigned the value zero.

err-label is the label of a branch target statement to which the program branches if there is an error condition in executing the CLOSE statement.

status is a CHARACTER expression that evaluates to either 'KEEP', 'DELETE', or 'FSYNC'.

Remarks

The STATUS= specifier determines the disposition of the file that is connected to the specified unit. 'KEEP' or 'FSYNC' shall not be specified for a file whose status prior to execution of a CLOSE statement is 'SCRATCH'. If 'KEEP' or 'FSYNC' is specified for a file that exists, the file continues to exist after the execution of a CLOSE statement. If 'KEEP' or 'FSYNC' is specified for a file that does not exist, the file will not exist after the execution of a CLOSE statement. If 'DELETE' is specified, the file will not exist after the execution of a CLOSE statement. If 'FSYNC'is specified, the file is synchronized. If STATUS= specifier is omitted, the default value is 'KEEP' , unless the file status prior to execution of the CLOSE statement is 'SCRATCH', in which case the default value is DELETE.

Example

```
close (8, status = 'keep')    ! unit 8 closed and kept
close (err = 200, unit = 9)   ! unit 9 closed; if error
                              ! occurs, branch to label
                              ! 200
```

2.72 CMPLX Intrinsic Function

Description

Convert to type COMPLEX.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
CMPLX	---	2 or 3	INTEGER or REAL or binary, octal, or hexadecimal constant [, INTEGER or REAL or binary, octal, or hexadecimal constant] , INTEGER	COMPLEX
	---	2	COMPLEX , INTEGER	COMPLEX
	---	1 or 2	INTEGER or REAL or binary, octal, or hexadecimal constant [, INTEGER or REAL or binary, octal, or hexadecimal constant]	Single-prec COMPLEX
	---	1	COMPLEX	Single-prec COMPLEX
DCMPLX	---	1 or 2	INTEGER or REAL [, INTEGER or REAL]	Double-prec COMPLEX

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	---	1	COMPLEX	Double-prec COMPLEX
QCMPLX	---	1 or 2	INTEGER or REAL [, INTEGER or REAL]	Quad-prec COMPLEX
	---	1	COMPLEX	Quad-prec COMPLEX

```
result = CMPLX(X[,Y][,KIND])
result = DCMPLX(X[,Y])
result = QCMPLX(X[,Y])
```

Required Argument(s)

X

X shall be of type REAL, INTEGER, COMPLEX, or binary, octal, or hexadecimal constant.

Optional Argument(s)

Y

Y shall be of type REAL, INTEGER, or binary, octal, or hexadecimal constant. If *X* is of type COMPLEX, *Y* shall not be present.

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The value of the result is the COMPLEX number whose real part has the value of *X*, if *X* is an INTEGER or a REAL; whose real part has the value of the real part of *X*, if *X* is of type COMPLEX; and whose imaginary part has the value of *Y*, if present, and zero otherwise.

For CMPLX, the result is of type COMPLEX. If *KIND* is present the result is of kind *KIND*; otherwise, it is of default kind.

For DCMPLX, the result is of type double precision COMPLEX.

For QCMPLX, the result is of type quadruple precision COMPLEX.

Remarks

CMPLX, DCMPLX, and QCMPLX convert to COMPLEX type.

For CMPLX, if *Y* is absent and *X* is not COMPLEX, it is as if *Y* were present with the value REAL(0.0 [, *KIND*]). If *Y* is absent and *X* is COMPLEX, it is as of *Y* were present with the value AIMAG(*X*). CMPLX(*X* , *Y* [, *KIND*]) has the complex value whose real part is REAL(*X* [, *KIND*]) and whose imaginary part is REAL(*Y* [, *KIND*]).

For DCMPLX, if *Y* is absent and *X* is not COMPLEX, it is as if *Y* were present with the value 0.0D0. If *Y* is absent and *X* is COMPLEX, it is as of *Y* were present with the value DIMAG(*X*). DCMPLX(*X* , *Y*) has the COMPLEX value whose real part is DBLE(*X*) and whose imaginary part is DBLE(*Y*).

For QCMPLX, if *Y* is absent and *X* is not COMPLEX, it is as if *Y* were present with the value 0.0Q0. If *Y* is absent and *X* is COMPLEX, it is as of *Y* were present with the value QIMAG(*X*). QCMPLX(*X* , *Y*) has the COMPLEX value whose real part is QEXT(*X*) and whose imaginary part is QEXT(*Y*).

Example

```
complex :: y,z
y = cmplx (3.2, 4.7)  ! y is assigned (3.2, 4.7)
z = cmplx (3.2)       ! z is assigned (3.2, 0.0)
```

2.73 CODIMENSION Statement

Description

The CODIMENSION statement specifies a coarray.

Syntax

```
CODIMENSION [ :: ] coarray-decl-list
```

Where:

coarray-decl is

```
coarray-name  left-square-bracket  coarray-spec  right-square-bracket
```

coarray-name is the name of coarray.

See "1.17.1 Coarray Specifier" for *coarray-spec.*

left-square-bracket is a left-side square bracket. The left-side square bracket is '['.

right-square-bracket is a right-side square bracket. The right-side square bracket is ']'.

Example

```
codimension :: a[*] , b[2,*]
```

2.74 COMMAND_ARGUMENT_COUNT Intrinsic Function

Description

The Number of command arguments.

Class

Inquiry function.

Syntax

```
result = COMMAND_ARGUMENT_COUNT ( )
```

Result

The result is of type default INTEGER scalar. Its value is the number of specified command arguments.

Remarks

If there is no specified command argument, the result value is zero. The command name is not included in a number of command arguments.

Example

```
i = command_argument_count ()  ! % a.out arg1 arg2 arg3
                               ! The variable "i" is three that is
                               ! the number of command arguments.
```

2.75 COMMON Statement

Description

The COMMON statement provides a global data facility. It specifies blocks of physical storage, called common blocks that can be accessed by any scoping unit in an executable program.

Syntax

```
COMMON [ / [ common-block-name ] / ] common-block-object-list   &
&   [ [ , ] / [ common-block-name ] / common-block-object-list ] ...
```

Where:

common-block-name is the name of a common block being declared.

common-block-object-list is a comma-separated list of common block objects.

common-block-object is

```
variable-name [ ( explicit-shape-spec-list ) ]        or
proc-pointer-name
```

variable-name is a name of data object that is declared to be in the common block.

common-block-object shall not be a dummy argument, an allocatable variable, an object of a derived type containing an allocatable variable as an ultimate component, an automatic object, a function name, an entry name, a variable with BIND attribute, a result name, a coarray, or a name made accessible by use association.

explicit-shape-spec-list is a comma-separated list of explicit shape specifier.

explicit-shape-spec is

```
[ lower-bound : ] upper-bound
```

lower-bound is a constant specification expression. If the *lower-bound* is omitted, the default value is 1.

upper-bound is a constant specification expression.

If a *variable-name* appears with an *explicit-shape-spec-list*, it declares to have the DIMENSION attribute and specifies the array properties that apply, and such *variable-name* shall not have the POINTER attribute.

If a *common-block-object* is of a derived-type, it shall be a sequence type or it shall have a BIND attribute,and must not have default initialization.

Remarks

In each COMMON statement, the data objects whose names appear in a common block object list following a common block name are declared to be in that common block. If the first common block name is omitted, all data objects whose names appear in the first common block object list are specified to be in blank common. Alternatively, the appearance of two slashes with no common block name between them declares the data objects whose names appear in the common block object list that follows to be in blank common.

A common block name or blank common can appear multiple times in one or more COMMON statements in a scoping unit. In such case, the *common-object-list* is treated as a continuation of the *common-object-list* for that common block.

For each common block, a storage sequence is formed of storage sequences of all data objects in the common block, in the order they appear in *common-object-list*s in the scoping unit. If any storage sequence is associated by equivalence association with the storage sequence of the common block, the sequence can be extended only by adding storage units beyond the last storage unit.

Within a program, the storage sequences of all common blocks with the same name (or all blank commons) have the same first storage unit. This results in the association of objects in different scoping units.

A nonpointer object of default INTEGER type, default REAL type, double precision REAL type, default COMPLEX type, default LOGICAL type, or numeric sequence type shall become associated only with nonpointer objects of these types.

A nonpointer object of type default CHARACTER or character sequence type shall become associated only with nonpointer objects of these types.

A nonpointer object of a derived type that is not a numeric sequence or character sequence type shall become associated only with nonpointer objects of the same type.

A nonpointer object of intrinsic type other than default INTEGER, default REAL, double precision REAL, default COMPLEX, default LOGICAL, or default CHARACTER shall become associated only with nonpointer objects of the same type and type parameters.

A pointer shall become storage associated only with pointers of the same type, type parameters, and rank.

A blank common has the same properties as a named common, except:

1. Execution of a RETURN or END statement may cause data objects in a named common block to become undefined unless the common block name has been declared in a SAVE statement, but never causes data objects in blank common to become undefined.

2. Named common blocks of the same name shall be the same size in all scoping units of a program in which they appear, but blank common blocks may be of different size.
3. A data object in a named common block may be initially defined in a DATA or type declaration statement in a block data program unit, but objects in blank common shall not be initially defined.

An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to be associated.

An EQUIVALENCE statement shall not cause storage units to be added before the first storage unit of the common block.

Example

```
common /first/ a,b,c        ! a, b, and c are in named
                            ! common first
common d,e,f, /second/ g    ! d, e, and f are in blank
                            ! common, g is in named
                            ! common second
common /first/ h            ! h is also in first
```

2.76 COMPILER_OPTIONS Intrinsic Module Function

Description

Effective compiler options are returned at compilation time.

As for returned options information, see "Fortran User's Guide".

Class

Inquiry function.

Syntax

```
result = COMPILER_OPTIONS (  )
```

Result

Default CHARACTER scalar.

Example

```
use,intrinsic::iso_fortran_env,only:compiler_options
print *,compiler_options()
end
```

2.77 COMPILER_VERSION Intrinsic Module Function

Description

Version information on the compiler used is returned.

As for returned options information, see "Fortran User's Guide".

Class

Inquiry function.

Syntax

```
result = COMPILER_VERSION (  )
```

Result

Default CHARACTER scalar.

Example

```
use,intrinsic::iso_fortran_env,only:compiler_version
print *,compiler_version()
end
```

2.78 COMPLEX Type Declaration Statement

Description

The COMPLEX type declaration statement declares entities of type COMPLEX. See "2.469 Type Declaration Statement" for type declaration statement.

Syntax

```
COMPLEX [ kind-selector ] [ [ , attr-spec ] ... :: ] entity-decl-list
```

2.79 Computed GO TO Statement (obsolescent feature)

Description

The computed GO TO statement causes transfer of control to one of a list of labeled statements.

Syntax

```
GO TO ( label-list ) [ , ] scalar-int-expr
```

Where:

label-list is a comma-separated list of labels. Each label in *label-list* shall be the label of a branch target statement in the current scoping unit.

The same statement label may appear more than once in a *label-list*.

scalar-int-expr is a scalar INTEGER expression.

Remarks

Execution of a computed GO TO statement causes evaluation of *scalar-int-expr*. If this value is *i* such that 1 <= *i* <= *n* , where *n* is the number of labels in *labels*, a transfer of control occurs so that the next statement executed is the one identified by the *i*th label in *labels*. If *i* is less than 1 or greater than *n*, the execution sequence continues as though a CONTINUE statement were executed.

Example

```
      go to (10,20,30) i
10    a = a+1  ! if i=1 control transfers here
20    a = a+1  ! if i=2 control transfers here
30    a = a+1  ! if i=3 control transfers here
```

2.80 CONJG Intrinsic Function

Description

Conjugate of a complex number.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
CONJG	---	1	COMPLEX	COMPLEX
	CONJG		Single-prec COMPLEX	Single-prec COMPLEX
	DCONJG		Double-prec COMPLEX	Double-prec COMPLEX
	QCONJG		Quad-prec COMPLEX	Quad-prec COMPLEX

```
result = CONJG ( Z )
```

Argument(s)

Z

Z shall be of type COMPLEX.

Result

The result is of type COMPLEX and of the same kind as *Z*. Its value is the same as that of *Z* with the imaginary part negated.

Remarks

CONJG, DCONJG, and QCONJG form the conjugate of a complex number.
The result has the value CMPLX(REAL(*Z*), -AIMAG(*Z*)).

The generic name, CONJG, may be used with any COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
complex :: x
x = conjg ((2.1, -3.2))  ! x is assigned the value (2.1, 3.2)
```

2.81 CONTAINS Statement

Description

The CONTAINS statement separates the body of a main program, module, or subprogram from any internal or module subprograms it contains.

Syntax

```
CONTAINS
```

Remarks

The CONTAINS statement is nonexecutable statement.

The CONTAINS statement is not allowed in a block data program unit and an internal subprogram.

Example

```
subroutine outside (a)
  implicit none
  real, intent(in) :: a
  integer :: i, j
  real :: x
  ...
  call inside (i)
  x = sin (3.89)        ! not the intrinsic sin()
  ...
  contains
  subroutine inside (k) ! not available outside outside()
    implicit none
    integer, intent(in) :: k
    ...
  end subroutine inside
  function sin (m)      ! not available outside outside()
    implicit none
    real :: sin
    real, intent(in) :: m
    ...
  end function sin
end subroutine outside
```

2.82 CONTIGUOUS Statement

Description

The CONTIGUOUS attribute can specify for only assumed-shape array or an array pointer:

- If the CONTIGUOUS attribute is specified with an assumed-shape array, the assumed-shape array can only be argument associated with a contiguous effective argument.
- If the CONTIGUOUS attribute is specified with an array pointer, the array pointer can only be pointer associated with a contiguous target.

Syntax

```
CONTIGUOUS [ :: ] object-name-list
```

Where:

object-name-list is a comma-separated list of object name.

The CONTIGUOUS statement specifies the CONTIGUOUS attribute for a list of objects. The *object-name* shall be an assumed-shape array or an array pointer.

Remarks

An object with nonzero-size array and nonzero-length array is CONTIGUOUS under the following condition:

- An object with the CONTIGUOUS attribute. or,
- A nonpointer whole array that is not assumed-shape. or,
- An assumed-shape array that is argument associated with an array that is contiguous. or,
- An array allocated by an ALLOCATE statement. or,
- A pointer associated with a contiguous target. or,
- An array section under the following conditions:
  - The entity is contiguous. and,
  - It does not have a vector subscript. and,
  - It has a consecutive subset. and,
  - If a substring range appears, the substring range specifies all of the characters type element. and,
  - If a structure component appears, the rightmost is an array.

Example

```
integer,pointer,dimension(:,:)::c
contiguous::c
```

2.82.1 Simply CONTIGUOUS

An array is a simply contiguous under the following conditions:

- If the array is an array section, a vector subscript does not appear. and,
- If the array is an array section, all subscript triplets are colons excluding the last subscript triplet. and,
- If the array is an array section, a stride does not appear in the last subscript triplet. and,
- If the array is an array section, no subscript triplet is preceded by a section subscript that is a subscript of scalar integer expression. and,
- A substring does not appear. and,
- If the array is a structure component, the rightmost part name is array. and,

- Under the following conditions:
  - The object has a CONTIGUOUS attribute. or,
  - The array entity is not an array pointer or an assumed shape array.

Example

```
integer,dimension(5,3)::array
...
array(2:4,3)      ! Simply CONTIGUOUS
array(:,2)        ! Simply CONTIGUOUS
```

2.83 CONTINUE Statement

Description

Execution of a CONTINUE statement has no effect.

Syntax

CONTINUE

Example

```
      do 10 i=1,100
        a(i) = b(i,i)
10    continue
```

2.84 COS Intrinsic Function

Description

Cosine of an argument in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
COS	---	1	REAL or COMPLEX	REAL or COMPLEX
	COS		Single-prec REAL	Single-prec REAL
	DCOS		Double-prec REAL	Double-prec REAL
	QCOS		Quad-prec REAL	Quad-prec REAL
	CCOS		Single-prec COMPLEX	Single-prec COMPLEX
	CDCOS		Double-prec COMPLEX	Double-prec COMPLEX
	CQCOS		Quad-prec COMPLEX	Quad-prec COMPLEX

result = COS (*X*)

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the cosine of *X*.

Remarks

COS, DCOS, QCOS, CCOS, CDCOS, and CQCOS evaluate the cosine of a REAL or COMPLEX data in radians.

For a REAL(KIND=4) argument, the domain shall be ABS(*X*) < 8.23E+05 .

For a REAL(KIND=8) argument, the domain shall be DABS(*X*) < 3.53D+15 .

For a REAL(KIND=16) argument, the domain shall be QABS(*X*) < $2.0Q0^{62} * \pi$.

For a COMPLEX(KIND=4) argument, the domain shall be ABS(REAL(*X*)) < 8.23E+05 and ABS(AIMAG(*X*)) < 89.415E0 .

For a COMPLEX(KIND=8) argument, the domain shall be DABS(DREAL(*X*)) < 3.53D+15 and DABS(DIMAG(*X*)) < 710.475D0 .

For a COMPLEX(KIND=16) argument, the domain shall be QABS(QREAL(*X*)) < $2.0Q0^{62} * \pi$ and QABS(QIMAG(*X*)) < 11357.125Q0 .

The generic name, COS, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = cos(.5)
```

2.85 COSD Intrinsic Function

Description

Cosine of an argument in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
COSD	---	1	REAL	REAL
	COSD		Single-prec REAL	Single-prec REAL
	DCOSD		Double-prec REAL	Double-prec REAL
	QCOSD		Quad-prec REAL	Quad-prec REAL

```
result = COSD ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the REAL type and kind as *X*. Its value is a REAL representation of the cosine of *X*.

Remarks

COSD, DCOSD, and QCOSD evaluate the cosine of a real data in degrees. The result has the value COS($\pi / 180 * X$) .

For a REAL(KIND=4) argument, the domain shall be ABS(*X*) < 4.72E+07 .

For a REAL(KIND=8) argument, the domain shall be DABS(*X*) < 2.03D+17 .

For a REAL(KIND=16) argument, the domain shall be QABS(*X*) < $2.0Q0^{62}$ * 180 .

The generic name, COSD, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = cosd(.5)
```

2.86 COSH Intrinsic Function

Description

Hyperbolic cosine.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
COSH	---	1	REAL or COMPLEX	REAL or COMPLEX
	COSH		Single-prec REAL	Single-prec REAL
	DCOSH		Double-prec REAL	Double-prec REAL
	QCOSH		Quad-prec REAL	Quad-prec REAL

```
result = COSH ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the hyperbolic cosine of *X*.

Remarks

COSH evaluates the hyperbolic cosine of REAL or COMPLEX data. DCOSH and QCOSH evaluate the hyperbolic cosine of a REAL data.

For a REAL(KIND=4) argument, the domain shall be ABS(*X*) < 89.415E0 .

For a REAL(KIND=8) argument, the domain shall be DABS(*X*) < 710.475D0 .

For a REAL(KIND=16) argument, the domain shall be QABS(*X*) < 11357.125Q0 .

For a COMPLEX(KIND=4) argument, the domain shall be ABS(REAL(*X*)) < 89.415E0 and ABS(AIMAG(*X*)) < 8.23E+05 .

For a COMPLEX(KIND=8) argument, the domain shall be DABS(DREAL(*X*)) < 710.475D0 and DABS(DIMAG(*X*)) < 3.53D+15 .

For a COMPLEX(KIND=16) argument, the domain shall be QABS(QREAL(*X*)) < 11357.125Q0 and QABS(QIMAG(*X*)) < $2.0Q0^{62} * \pi$.

The generic name, COSH, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = cosh(.5)
```

2.87 COSQ Intrinsic Function

Description

Cosine of an argument in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
COSQ	---	1	REAL or COMPLEX	REAL or COMPLEX
	COSQ		Single-prec REAL	Single-prec REAL
	DCOSQ		Double-prec REAL	Double-prec REAL
	QCOSQ		Quad-prec REAL	Quad-prec REAL
	CCOSQ		Single-prec COMPLEX	Single-prec COMPLEX
	CDCOSQ		Double-prec COMPLEX	Double-prec COMPLEX
	CQCOSQ		Quad-prec COMPLEX	Quad-prec COMPLEX

```
result = COSQ ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the cosine of *X*.

Remarks

COSQ, DCOSQ, QCOSQ, CCOSQ, CDCOSQ, and CQCOSQ evaluate the cosine of a real or complex data in quadrants. The result has the value COS($\pi / 2 * X$) .

For a COMPLEX(KIND=4) argument, the domain shall be ABS(REAL(*X*)) < 5.24E+05 and ABS(AIMAG(*X*)) < 56.92E0 .

For a COMPLEX(KIND=8) argument, the domain shall be DABS(DREAL(*X*)) < 2.25D+15 and DABS(DIMAG(*X*)) < 452.305D0 .

For a COMPLEX(KIND=16) argument, the domain shall be QABS(QREAL(*X*)) < $2.0Q0^{63}$ and QABS(QIMAG(*X*)) < 7230.125Q0 .

The generic name, COSQ, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = cosq(.5)
```

2.88 COTAN Intrinsic Function

Description

Cotangent of an argument in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
COTAN	---	1	REAL	REAL
	COTAN		Single-prec REAL	Single-prec REAL
	DCOTAN		Double-prec REAL	Double-prec REAL
	QCOTAN		Quad-prec REAL	Quad-prec REAL

```
result = COTAN ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is a REAL representation of the cotangent of *X*.

Remarks

COTAN, DCOTAN, and QCOTAN evaluate the cotangent of a REAL data in radians.

For a REAL(KIND=4) argument, the domain shall be ABS(*X*) < 8.23E+05 .

For a REAL(KIND=8) argument, the domain shall be DABS(*X*) < 3.53D+15 .

For a REAL(KIND=16) argument, the domain shall be QABS(*X*) < $2.0Q0^{62} * \pi$.

The generic name, COTAN, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = cotan(.5)
```

2.89 COTAND Intrinsic Function

Description

Cotangent of an argument in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
COTAND	---	1	REAL	REAL
	COTAND		Single-prec REAL	Single-prec REAL
	DCOTAND		Double-prec REAL	Double-prec REAL
	QCOTAND		Quad-prec REAL	Quad-prec REAL

```
result = COTAND ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is a REAL representation of the cotangent of *X*.

Remarks

COTAND, DCOTAND, and QCOTAND evaluate the cotangent of a REAL data in degrees.
The result has the value COTAN($\pi / 180 * X$) .

For a REAL(KIND=4) argument, the domain shall be ABS(*X*) < 4.72E+07 .

For a REAL(KIND=8) argument, the domain shall be DABS(*X*) < 2.03D+17 .

For a REAL(KIND=16) argument, the domain shall be QABS(*X*) < $2.0Q0^{62} * 180$.

The generic name, COTAND, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = cotand(.5)
```

2.90 COTANQ Intrinsic Function

Description

Cotangent of an argument in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
COTANQ	---	1	REAL	REAL
	COTANQ		Single-prec REAL	Single-prec REAL
	DCOTANQ		Double-prec REAL	Double-prec REAL
	QCOTANQ		Quad-prec REAL	Quad-prec REAL

```
result = COTANQ ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is a REAL representation of the cotangent of *X*.

Remarks

COTANQ, DCOTANQ, and QCOTANQ evaluate the cotangent of a REAL data in quadrants.
The result has the value COTAN($\pi / 2 * X$) .

The generic name, COTANQ, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = cotanq(.5)
```

2.91 COUNT Intrinsic Function

Description

Count the number of true elements in a mask along a given dimension.

Class

Transformational function.

Syntax

```
result = COUNT ( MASK [ , DIM , KIND] )
```

Required Argument(s)

MASK

MASK shall be of type LOGICAL. It shall not be a scalar.

Optional Argument(s)

DIM

DIM shall be a scalar of type INTEGER with a value within the range 1 <= *DIM* < *n* , where *n* is the rank of *MASK*. The corresponding actual argument shall not be an optional dummy argument.

KIND

KIND shall be a scalar of type INTEGER initialization expression.

Result

Its value and rank are computed as follows:

1. If *DIM* is absent or *MASK* has rank one, the result is scalar. The result is the number of elements for which *MASK* is true.
2. If *DIM* is present or *MASK* has rank two or greater, the result is an array of rank *n*-1 and of shape $(d_1,d_2,...,d_{DIM-1},d_{DIM+1},...,d_n)$ where $(d_1,d_2,...,d_n)$ is the shape of *MASK* and *n* is the rank of *MASK*. The result is the number of true elements for each corresponding vector in *MASK*.

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

Example

```
integer, dimension (2,3) :: a, b
integer, dimension (2) :: c
integer, dimension (3) :: d
integer :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                  !  represents  |1 3 5|
                  !              |2 4 6|
b = reshape((/1,2,3,5,6,4/), (/2,3/))
                  !  represents  |1 3 6|
                  !              |2 5 4|
e = count(a==b)   ! e is assigned the value 3
d = count(a==b, 1)! d is assigned the value 2,1,0
c = count(a==b, 2)! c is assigned the value 2,1
```

2.92 CO_MAX Intrinsic Subroutine

Description

Calculates maximum value on each image elementally.

The intrinsic subroutine must be called as the same statement on all active images and must appear only in a context that allows an image control statement.

Class

Collective subroutine.

Syntax

```
CALL CO_MAX ( A [, RESULT_IMAGE, STAT, ERRMSG ] )
```

Required Argument(s)

A

A must be a coarray of type integer, real, or character. It must have the same type and type parameters on each image. It is an INTENT(INOUT) argument. If it is a scalar, it is defined with the maximum value of *A* on each image. If *A* is an array, each element of *A* is defined with the maximum value of that element on each image. It must have the same shape on each image if *A* is an array. If *RESULT_IMAGE* is specified, the computed value is assigned to *A* on image *RESULT_IMAGE*, and *A* on all other images becomes undefined. If *RESULT_IMAGE* is not specified, the computed value is assigned *A* on each image.

Optional Argument(s)

RESULT_IMAGE

It must be an integer scalar. It is an INTENT(IN) argument. It must have the same value on each image.

STAT

STAT must be a noncoindexed scalar integer. It is an INTENT(OUT) argument. If the execution terminates successfully, 0 is assigned to the variable. If error occurs during the execution, either of following value is assigned:

- The value of STAT_STOPPED_IMAGE defined in ISO_FORTRAN_ENV intrinsic module.
- Diagnostic message number that comes during execution.

ERRMSG

ERRMSG must be a noncoindexed default character scalar. It is an INTENT(INOUT) argument.

Example

```
integer, save :: k[*]
k = this_image()
sync all
call co_max(k) ! maximum image index is assigned to variable k in each image
```

2.93 CO_MIN Intrinsic Subroutine

Description

Calculates minimum value on each image elementally.

The intrinsic subroutine must be called as the same statement on all active images and must appear only in a context that allows an image control statement.

Class

Collective subroutine.

Syntax

```
CALL CO_MIN ( A [, RESULT_IMAGE, STAT, ERRMSG ] )
```

Required Argument(s)

A

A must be a coarray of type integer, real, or character. It must have the same type and type parameters on each image. It is an INTENT(INOUT) argument. If *A* is a scalar, *A* is defined with the minimum value of *A* on each image. If *A* is an array, each element of *A* becomes defined with the minimum value of that element on each image. It must have the same shape on each image if *A* is an array. If *RESULT_IMAGE* is specified, the computed value is assigned to *A* on image *RESULT_IMAGE*, and *A* on all other images becomes undefined. If *RESULT_IMAGE* is not specified, the computed value is assigned *A* on each image.

Optional Argument(s)

RESULT_IMAGE

It must be an integer scalar. It is an INTENT(IN) argument. It must have the same value on each image.

STAT

STAT must be a noncoindexed scalar integer. It is an INTENT(OUT) argument. If the execution terminates successfully, 0 is assigned to the variable. If error occurs during the execution, either of following value is assigned:

- The value of STAT_STOPPED_IMAGE defined in ISO_FORTRAN_ENV intrinsic module.
- Diagnostic message number that comes during execution.

ERRMSG

ERRMSG must be a noncoindexed default character scalar. It is an INTENT(INOUT) argument.

Example

```
integer, save :: k[*]
k = this_image()
sync all
call co_min(k) ! minimum image index 1 is assigned to variable k on each image
```

2.94 CO_SUM Intrinsic Subroutine

Description

Sums value on each image elementally.

The intrinsic subroutine must be called as the same statement on all active images and must appear only in a context that allows an image control statement.

Class

Collective subroutine.

Syntax

```
CALL CO_SUM ( A [, RESULT_IMAGE, STAT, ERRMSG ] )
```

Required Argument(s)

A

A must be a coarray of numeric type. It must have the same type and type parameters on each image. It is an INTENT(INOUT) argument. If *A* is a scalar, *A* is defined with the sum of the value of *A* on each image. If *A* is an array, each element of *A* is defined with the sum of the value of that element on each image. It must have the same shape on each image if *A* is an array. If *RESULT_IMAGE* is specified, the computed value is assigned to *A* on image *RESULT_IMAGE*, and *A* on all other images becomes undefined. If *RESULT_IMAGE* is not specified, the computed value is assigned *A* on each image.

Optional Argument(s)

RESULT_IMAGE

It must be an integer scalar. It is an INTENT(IN) argument. It must have the same value on each image.

STAT

STAT must be a noncoindexed scalar integer. It is an INTENT(OUT) argument. If the execution terminates successfully, 0 is assigned to the variable. If error occurs during the execution, either of following value is assigned:

- The value of STAT_STOPPED_IMAGE defined in ISO_FORTRAN_ENV intrinsic module.
- Diagnostic message number that comes during execution.

ERRMSG

ERRMSG must be a noncoindexed default character scalar. It is an INTENT(INOUT) argument.

Example

```
integer, save :: k[*]
k = this_image()
sync all
call co_sum(k) ! "1 + ... + max image index" is assigned to variable k on each image
```

2.95 CPU_TIME Intrinsic Subroutine

Description

Processor Time.

Class

Subroutine.

Syntax

```
CALL  CPU_TIME ( TIME )
```

Argument(s)

TIME

TIME shall be a scalar REAL. It is an INTENT(OUT) argument that is assigned the processor time in seconds.

When the system cannot acquire the processor time, -1.0 is set.

Example

```
call cpu_time(start_time)
x = cos(2.0)
call cpu_time(end_time)
cos_time = end_time - start_time
           ! time to calculate and store the cosine of 2.0
```

2.96 CRITICAL Construct

Description

CRITICAL construct prevents more than one image from executing a block at a same time.

Syntax

```
[ critical-construct-name : ] CRITICAL
 block
END CRITICAL [ critical-construct-name ]
```

Where:

critical-construct-name is an optional name given to the CRITICAL construct. If the CRITICAL statement is identified by a *critical-construct-name*, the corresponding END CRITICAL statement must specify the same *critical-construct-name*. If the CRITICAL statement is not identified by a *critical-construct-name*, the corresponding END CRITICAL statement must not specify a *critical-construct-name*.

block is a sequence of zero or more statements or executable constructs. It must not include RETURN statement (see "2.397 RETURN Statement") or image control statement (see "1.18 Image Control Statement").

Remarks

A branch in a CRITICAL construct must not have a branch target that is outside the CRITICAL construct. A branch that is outside the CRITICAL construct must not have a branch target in a CRITICAL construct.

Image control statement must not be executed by the procedure in CRITICAL construct.

Example

```
INTEGER, SAVE :: A[*]
A[1] = 0
K = THIS_IMAGE()
SYNC ALL
CRITICAL
  A[1] = A[1] + K  ! an image do not execute this statement while another image is executing this
END CRITICAL
```

2.97 CSHIFT Intrinsic Function

Description

Circular shift of all rank one sections in an array. Elements shifted out at one end are shifted in at the other. Different sections can be shifted by different amounts and in different directions by using an array-valued shift.

Class

Transformational function.

Syntax

```
result = CSHIFT ( ARRAY , SHIFT [ , DIM ] )
```

Required Argument(s)

ARRAY

ARRAY can be of any type. It shall not be scalar.

SHIFT

SHIFT shall be of type INTEGER and shall be scalar if *ARRAY* is of rank one; otherwise it shall be scalar or of rank *n*-1 and of shape $(d_1,d_2,...,d_{DIM-1},d_{DIM+1},...,d_n)$, where $(d_1,d_2,...,d_n)$, is the shape of *ARRAY*.

Optional Argument(s)

DIM

DIM shall be a scalar INTEGER with a value in the range 1 <= *DIM* <= *n* , where *n* is the rank of *ARRAY*. If *DIM* is omitted, it is as if it were present with the value one.

Result

The result is of the same type, kind, and shape as *ARRAY*.

If *ARRAY* is of rank one, the value of the result is the value of *ARRAY* circularly shifted *SHIFT* elements. A *SHIFT* of *n* performed on *ARRAY* gives a result value of *ARRAY*(1+MODULO(*i*+*SHIFT*-1, SIZE(*ARRAY*))) for element *i*.

If *ARRAY* is of rank two or greater, each complete vector along dimension *DIM* is circularly shifted *SHIFT* elements. *SHIFT* can be array-valued.

Example

```
integer, dimension (2,3) :: a, b
integer, dimension (3) :: c, d
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                  !  represents  |1 3 5|
                  !              |2 4 6|
c = (/1,2,3/)
b = cshift(a,1)   ! b is assigned the value |2 4 6|
                  !                         |1 3 5|
b = cshift(a,-1,2)! b is assigned the value |5 1 3|
                  !                         |6 2 4|
b = cshift(a,c,1) ! b is assigned the value |2 3 6|
                  !                         |1 4 5|
d = cshift(c,2)   ! d is assigned the value |3 1 2|
```

2.98 CTIME Service Function

Description

Returns the system time as an ASCII string.

Syntax

```
ch = CTIME ( time )
```

Argument(s)

time

Default INTEGER scalar. System time.

Result

Default CHARACTER scalar. Returns a 24-byte character string.

Example

```
use service_routines,only:ctime,time
character(len=24) :: ch
ch = ctime(time())
end
```

2.99 CYCLE Statement

Description

The CYCLE statement curtails the execution of a single iteration of a DO loop.

Syntax

```
CYCLE [ do-construct-name ]
```

Where:

do-construct-name is the name of a DO construct that contains the CYCLE statement.

A CYCLE statement shall appear in a DO construct.

Execution of a CYCLE statement causes to curtail the execution of a single iteration of the DO loop that identified by *do-construct-name*, if any. Otherwise, the execution of a single iteration of the innermost DO construct in which it appears is curtailed.

Example

```
outer: do i=1, 10
inner:   do j=1, 10
           if (i>ii) cycle outer
           if (j>jj) cycle ! cycles to inner
           ...
         end do inner
       end do outer
```

2.100 C_ASSOCIATED Intrinsic Module Function

Description

The C_ASSOCIATED intrinsic module function determines the associate status of C_PTR type and C_FUNPTR type pointers.

This function provides the associate status of *C_PTR_1* , or indicates whether or not *C_PTR_1* and *C_PTR_2* are associated with the same entity.

Class

Inquiry function.

Syntax

```
result = C_ASSOCIATED ( C_PTR_1 [ , C_PTR_2 ] )
```

Required Argument(s)

C_PTR_1

Must be the C_PTR type or C_FUNPTR type scalar.

Optional Argument(s)

C_PTR_2

Must be the same type scalar as *C_PTR_1.*

Result

- If *C_PTR_2* is omitted:
  - The result is false when *C_PTR_1* is a C language NULL pointer.

- The result is true in other cases.

- If *C_PTR_2* is present:
  - The result is false when *C_PTR_1* is a C language NULL pointer.
  - In other cases, the result is true if *C_PTR_1* equals *C_PTR_2,* but false if they are not equal.

Example

```
use , intrinsic :: iso_c_binding,only:c_funptr,c_funloc,c_f_procpointer
interface
  function ifun()  bind(c)
  end function
end interface
type (c_funptr)          :: cfunptr
procedure(ifun), pointer :: pifun
cfunptr = c_funloc(ifun)
call c_f_procpointer(cfunptr, pifun)
```

2.101 C_FUNLOC Intrinsic Module Function

Description

Returns the C address of the argument.

Class

Inquiry function.

Syntax

result = C_FUNLOC (*X*)

Argument(s)

X

Must be either an interoperable procedure or a procedure pointer associated with an interoperable procedure.

Result

C_FUNPTR type scalar.

The result is the derived type having the procedure pointer component of an implicit interface. When the C_F_PROCPOINTER intrinsic module subroutine is invoked, this result can be used as the actual argument CPTR.

Example

```
use , intrinsic :: iso_c_binding,only:c_funptr,c_funloc
interface
  function ifun () bind(c)
  end function
end interface
type (c_funptr)     :: cfunptr
cfunptr = c_funloc(ifun)
```

2.102 C_F_POINTER Intrinsic Module Subroutine

Description

Generates a Fortran data pointer from a C pointer.

Class

Subroutine.

Syntax

```
CALL C_F_POINTER ( CPTR , FPTR [ , SHAPE ] )
```

Required Argument(s)

CPTR

Must be the C_PTR type scalar, and is the INTENT (IN) argument. The value must be one of the following:

- C address of an interoperable data entity
- Result of a reference to a C_LOC intrinsic module function with an argument that is not interoperable, the *CPTR* value cannot be the same as C address of a Fortran variable that does not have the TARGET attribute.

FPTR

Must be a pointer, and is the INTENT(OUT) argument.

- If the *CPTR* value is the C address of an interoperable data entity, *FPTR* must be a data pointer that has the type, interoperable type, and type parameter of the entity. In this case, *FPTR* associates the *CPTR* target with the pointer. If *FPTR* is an array, the shape is specified by *SHAPE*, and the lower bound is 1.
- If the *CPTR* value is the result of a reference to a C_LOC(X) intrinsic module function from argument X that is not interoperable, *FPTR* must be a non-polymorphic scalar pointer having the same type and type parameter as X. In this case, if X or the target(if X is a pointer), must not be deallocated. *FPTR* associates X or the target with the pointer.

Optional Argument(s)

SHAPE

An INTEGER type one-rank array, and is the INTENT(IN) argument. Unless *FPTR* is an array, *SHAPE* must specify. The size must equal the *FPTR* rank.

Example

```
use , intrinsic :: iso_c_binding,only:c_loc,c_ptr,c_f_pointer
type(c_ptr)     :: cptr
integer,pointer :: x,y
integer,target  :: t
t=1
x=>t
cptr=c_loc(x)
call c_f_pointer ( cptr,y)
```

2.103 C_F_PROCPOINTER Intrinsic Module Subroutine

Description

Associates a C function pointer with a procedure pointer.

Class

Subroutine.

Syntax

```
CALL C_F_PROCPOINTER ( CPTR , FPTR )
```

Argument(s)

CPTR

Must be a C_FUNPTR type scalar, and is the INTENT(IN) argument. The value must be the C address of an interoperable procedure.

FPTR

Must be a procedure pointer, and is the INTENT(OUT) argument. The interface for *FPTR* must be interoperable with the *CPTR* target. *FPTR* associates the *CPTR* target with the pointer.

Example

```
use , intrinsic :: iso_c_binding,only:c_funptr,c_funloc,c_f_procpointer
interface
  function ifun()  bind(c)
  end function
end interface
type (c_funptr)          :: cfunptr
procedure(ifun), pointer :: pifun
cfunptr = c_funloc(ifun)
call c_f_procpointer(cfunptr, pifun)
```

2.104 C_LOC Intrinsic Module Function

Description

Returns the C address of the argument.

Class

Inquiry function.

Syntax

result = C_LOC (*X*)

Argument(s)

X

If an interoperable type and a type parameter, and one of the following:

- Interoperable variable with the TARGET attribute
- An allocated allocatable variable that has the TARGET attribute
- Associated scalar pointer or array pointer

If a nonpolymorphic scalar or array, with no length type parameter, and be one of the following:

- A variable that is not a pointer that is not an allocate variable, with the TARGET attribute
- An allocated allocatable variable with the TARGET attribute
- Associated pointer

It must not be a zero-length. If it is an array, it must be CONTIGUOUS(see "2.82 CONTIGUOUS Statement") and have nonzero size.

Result

C_PTR type scalar.

If *X* is a scalar type data entity, the result is the derived type of the scalar pointer entity where the C_PTR type matches the type and type parameter of *X*. If *X* is an array data entity, the result is the derived type of the scalar pointer entity where the C_PTR type matches the type and type parameter of *X*. If *X* is interoperable or a data entity with an interoperable type or type parameter, the result is that C address.

Example

```
use , intrinsic :: iso_c_binding,only:c_ptr,c_loc
integer(kind=4), target :: t
type(c_ptr) :: cptr
cptr = c_loc(t)
```

2.105 C_SIZEOF Intrinsic Module Function

Description

The number of bytes of the argument.

Class

Inquiry function.

Syntax

```
result = C_SIZEOF ( X )
```

Argument(s)

X

Must be an interoperable data that is not an assumed-size array.

Result

Eight-byte INTEGER. It is a scalar.

If the argument is scalar, size of the object is returned. If the argument is array, a result of the multiplication of the size of one element and the number of elements is returned.

Example

```
use,intrinsic::iso_c_binding,only:c_sizeof,c_int,c_size_t
integer(c_int),dimension(10)::i
integer(c_size_t)::k
k = c_sizeof(i)         ! k is assigned the value 40
print *,k
end
```

2.106 DATA Statement

Description

The DATA statement provides initial values for variables.

Syntax

```
DATA data-stmt-set [ [ , ] data-stmt-set ] ...
```

Where:

data-stmt-set is

```
data-stmt-object-list / data-stmt-value-list /
```

data-stmt-object-list is a comma-separated list of

```
variable            or
data-implied-do
```

variable is a variable name.

data-implied-do is

```
(data-i-do-object-list,data-i-do-variable=scalar-int-expr,scalar-int-expr[,scalar-int-expr])
```

data-i-do-object-list is a comma-separated list of

```
array-element                 or
scalar-structure-component    or
data-implied-do
```

array-element is an array element.

scalar-structure-component is a scalar structure component.

data-i-do-variable is a named scalar INTEGER variable.

scalar-int-expr is a scalar INTEGER expression that involves as primaries only constants, subobjects of constants, or DO variables of the containing *data-implied-do*s.

data-stmt-value-list is a comma-separated list of

```
[ data-stmt-repeat * ] data-stmt-constant
```

data-stmt-repeat is

```
scalar-int-constant              or
scalar-int-constant-subobject
```

scalar-int-constant is a scalar INTEGER constant that shall be positive or zero.

scalar-int-constant-subobject is a scalar subobject of INTEGER constant that shall be positive or zero.

data-stmt-constant is

```
scalar-constant                or
scalar-constant-subobject      or
signed-int-literal-constant    or
signed-real-literal-constant   or
NULL( )                        or
structure-constructor
```

Remarks

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

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

Except for variables in named common blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement, and this may be confirmed by a SAVE statement or a type declaration statement containing the SAVE attribute.

A variable whose name or designator is included in a *data-stmt-object-list* or a *data-i-do-object-list* 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, an allocatable variable, or an object of a derived type containing an ultimate component that is an allocatable variable.

In a *variable* that is a *data-stmt-object*, any subobject, section subscript, substring starting point, and substring ending point shall be an initialization expression.

In an *array-element* or a scalar-structure-component that is a *data-i-do-object*, any subscript shall be an expression whose primaries are either constants, subobjects of constants, or DO variables of the containing *data-implied-do*s, and each operation shall be intrinsic.

The *data-object-list* is expanded to form a sequence of pointers and scalar variables, referred to as "sequence of variables" in subsequent text. A nonpointer array whose unqualified name appears in a *data-stmt-object-list* is equivalent to a complete sequence of its array elements in array element order. An array section is equivalent to the sequence of its array elements in array element order.
A *data-implied-do* is expanded to form a sequence of array elements and structure components, under the control, of the implied-DO variable, as in the DO construct.

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

A zero-sized array or an implied-DO list with an iteration count of zero contributes no variables to the expanded sequence of variables, but a zero-length scalar character variable does contribute a variable to the list. A *data-stmt-constant* with a repeat factor of zero contributes no *data-stmt-constant*s to the expanded sequence of scalar *data-stmt-constant*s.

The expanded sequences of variables and *data-stmt-constant*s are in one-to-one correspondence. Each *data-stmt-constant* specifies the initial value or NULL( ) for the corresponding variable. The lengths of the two expanded sequences shall be the same.

A *data-stmt-constant* other than NULL( ) shall be compatible with its corresponding variable according to the rules of intrinsic assignment, and the variable becomes initially defined with the *data-stmt-constant* in accordance with the rules of intrinsic assignment.

If a variable has the POINTER attribute, the corresponding *data-stmt-constant* shall be NULL( ), and the pointer has an initial association status of disassociated.

Example

```
real :: a
integer, dimension (-3:3) :: ary1
integer, dimension (100) :: ary2
data a /3.78/, ary1 /7 * 1/
                    ! assigns 3.78 to a and 1 to each
                    ! element of ary1
data (ary2(i), i=1,100) /100*6/
                    ! assigns 6 to each element
```

2.107 DATE Service Subroutine

Description

Returns string in the form *yy-mm-dd*, where *yy* is the last two digits of the current year, *mm* is the current month, and *dd* is the current day of the month.

Syntax

```
CALL  DATE ( ch )
```

Argument(s)

ch

Default CHARACTER scalar. Variable containing at least 8 bytes of storage.

If the length of argument *ch* is greater than 8, blank characters are supplied.

Example

```
use service_routines,only:date
character(len=10) :: dt
call date(dt)
write (6,fmt="(1x,a)") dt
end
```

2.108 DATE_AND_TIME Intrinsic Subroutine

Description

Date and real-time clock data.

Class

Subroutine.

Syntax

```
DATE_AND_TIME ( [ DATE , TIME , ZONE , VALUES ] )
```

Optional Argument(s)

DATE

DATE shall be scalar and of type default CHARACTER. It is an INTENT(OUT) argument. Its leftmost eight characters are set to a value of the form *ccyymmdd*, where *cc* is the century, *yy* the year within the century, *mm* the month within the year, and *dd* the day within the month. If there is no date available, they are set to blank.

TIME

TIME shall be scalar and of type default CHARACTER. It is an INTENT(OUT) argument. Its leftmost ten characters are set to 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, they are set to blank.

ZONE

ZONE shall be scalar and of type default CHARACTER. It is an INTENT(OUT) argument. Its leftmost five characters are set to a value of the form +-*hhmm*, where *hh* and *mm* are the time difference with respect to Coordinated Universal Time (UTC, also known as Greenwich Mean Time) in hours and parts of an hour expressed in minutes, respectively. If there is no clock available, they are set to blank.

VALUES

VALUES shall be of type default INTEGER and of rank one. It is an INTENT(OUT) argument. The values returned in *VALUES* are as follows:

VALUES (1) the year (for example, 1990), or `-huge(0)` if there is no date available.

VALUES (2) the month of the year, or `-huge(0)` if there is no date available.

VALUES (3) the day of the month, or `-huge(0)` if there is no date available.

VALUES (4) the time difference with respect to Coordinated Universal Time (UTC) in minutes, or `-huge(0)` if this information is not available.

VALUES (5) the hour of the day, in the range of 0 to 23, or `-huge(0)` if there is no clock.

VALUES (6) the minutes of the hour, in the range of 0 to 59, or `-huge(0)` if there is no clock.

VALUES (7) the seconds of the minute, in the range of 0 to 60, or `-huge(0)` if there is no clock.

VALUES (8) the milliseconds of the second, in the range of 0 to 999, or `-huge(0)` if there is no clock.

Example

```
! called in San Jose, CA on June 20, 2013
! at 22:20:02.5
integer :: dt(8)
character (len=10) :: time, date, zone
call date_and_time (date, time, zone, dt)
! date is assigned the value "20130620"
! time is assigned the value "222002.500"
! zone is assigned the value "-800"
! dt is assigned the value: 2013,6,20,
!                           -480,22,20,2,500.
```

2.109 DBLE Intrinsic Function

Description

Convert to double precision REAL type.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
DBLE	---	1	One-byte INTEGER	Double-prec REAL
	DFLOTI		Two-byte INTEGER	Double-prec REAL
	DFLOAT		Four-byte INTEGER	Double-prec REAL
	DFLOTJ		Four-byte INTEGER	Double-prec REAL
	---		Eight-byte INTEGER	Double-prec REAL

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	DBLE		Single-prec REAL or binary, octal, or hexadecimal constant	Double-prec REAL
	---		Double-prec REAL	Double-prec REAL
	DBLEQ		Quad-prec REAL	Double-prec REAL
	---		Single-prec COMPLEX	Double-prec REAL
	DREAL		Double-prec COMPLEX	Double-prec REAL
	---		Quad-prec COMPLEX	Double-prec REAL

```
result = DBLE ( A )
```

Argument(s)

A

A shall be of type INTEGER, REAL, COMPLEX, or a binary, octal, or hexadecimal constant.

Result

The result is of double precision REAL type. Its value is a double precision representation of *A*. If *A* is of type COMPLEX, the result is a double precision representation of the real part of *A*.

Remarks

DBLE, DFLOTI, DFLOAT, DFLOTJ, DBLEQ, and DREAL convert numeric data to double precision REAL type.

If *A* is of type COMPLEX, the result is as much precision of the significant part of the real part of *A* as a double precision REAL datum can contain. If *A* is a binary, octal, or hexadecimal constant, it returns the same bit value that a variable of double precision real type is expressed by a binary, octal, or hexadecimal constant. If *A* is of type REAL(KIND=16), the result is as much precision as a double precision REAL datum can contain.

The generic name, DBLE, may be used with any INTEGER, REAL, COMPLEX, or a binary, octal, or hexadecimal constant argument.

The type of the result of each function is double precision REAL.

Example

```
double precision :: d
d = dble (1)  ! d is assigned the value 1.0d0
```

2.110 DEALLOCATE Statement

Description

The DEALLOCATE statement deallocates allocatable variables and pointer targets and disassociates pointers.

Syntax

```
DEALLOCATE ( allocate-object-list [ , dealloc-opt-list ] )
```

Where:

allocate-object-list is a comma-separated list of

```
variable-name          or
structure-component
```

allocate-object is a pointer or allocatable variable.

dealloc-opt-list is

```
STAT = stat-variable        or
ERRMSG = errmsg-variable
```

stat-variable is a scalar INTEGER variable.

errmsg-variable is a scalar default CHARACTER variable.

Remarks

If the optional STAT= is present and the DEALLOCATE statement succeeds, *stat-variable* is assigned the value zero. If STAT= is present and the DEALLOCATE statement fails, *stat-variable* is assigned the number of the error message generated at runtime.

If an error condition occurs during execution of the ALLOCATE statement or the DEALLOCATE statement, the status is assigned to the *errmsg-variable*. If no error condition has occurred, the *errmsg-variable* value is not changed.

If an error condition occurs during execution of a DEALLOCATE statement that does not contain the STAT= specifier, the executable program is terminated.

Deallocating an allocatable variable that is not currently allocated causes an error condition in the DEALLOCATE statement. An allocatable variable with the TARGET attribute shall not be deallocated through an associated pointer. Deallocating an allocatable variable with the TARGET attribute causes the pointer association status of any pointer associated with it to become undefined.

If a pointer appears in a DEALLOCATE statement, its association status shall be defined. Deallocating a pointer that is disassociated or whose target was not created by an ALLOCATE statement causes an error condition in the DEALLOCATE statement. If a pointer is currently associated with an allocatable array, the pointer shall not be deallocated.

Deallocating a pointer target causes the pointer association status of any other pointer that is associated with the target or a portion of the target to become undefined.

When a DEALLOCATE statement is executed for which an *allocate-object* is a coarray, there is an implicit synchronization of all images. On each image, execution of the segment following the statement is delayed until all other images have executed the same statement.

If the coarray is a dummy argument, the actual argument must be the same coarray on every image.

There is also an implicit synchronization of all images in association with the deallocation of a coarray caused by the execution of a RETURN or END statement.

Example

```
deallocate (a, b, stat=s) ! causes a and b to be deallocated.
                          ! If successful, s is assigned 0
```

2.111 DIGITS Intrinsic Function

Description

Number of significant binary digits.

Class

Inquiry function.

Syntax

```
result = DIGITS ( X )
```

Argument(s)

X

X shall be of type INTEGER or REAL. It can be scalar or array-valued.

Result

The result is of type default INTEGER. Its value is the number of significant binary digits in *X*.

The result value is as follows:

Type of *X*	The result value
One-byte INTEGER	7
Two-byte INTEGER	15

Type of *X*	The result value
Four-byte INTEGER	31
Eight-byte INTEGER	63
Single precision REAL	24
Double precision REAL	53
Quadruple precision REAL	113

Example

```
real :: r
integer :: i
i = digits (r)  ! i is assigned the value 24
```

2.112 DIM Intrinsic Function

Description

The difference between two numbers if the difference is positive; zero otherwise.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
DIM	---	2	One-byte INTEGER , One-byte INTEGER	One-byte INTEGER
	I2DIM		Two-byte INTEGER , Two-byte INTEGER	Two-byte INTEGER
	IIDIM		Two-byte INTEGER , Two-byte INTEGER	Two-byte INTEGER
	IDIM		Four-byte INTEGER , Four-byte INTEGER	Four-byte INTEGER
	JIDIM		Four-byte INTEGER , Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER , Eight-byte INTEGER	Eight-byte INTEGER
	DIM		Single-prec REAL , Single-prec REAL	Single-prec REAL
	DDIM		Double-prec REAL , Double-prec REAL	Double-prec REAL
	QDIM		Quad-prec REAL , Quad-prec REAL	Quad-prec REAL

```
result = DIM ( X , Y )
```

Argument(s)

X

X shall be of type INTEGER or REAL.

Y

Y shall be of the same type and kind as *X*.

Result

The result is of the same type as *X*. Its value is *X* - *Y* if *X* is greater than *Y* and zero otherwise.

Remarks

The value of the result is *X* - *Y* if *X* > *Y*, and zero otherwise.

The generic name, DIM, may be used with any INTEGER or REAL argument.

The type of the result of each function is the same as the type of the arguments.

Example

```
z = dim(1.1, 0.8) ! z is assigned the value 0.3
z = dim(0.8, 1.1) ! z is assigned the value 0.0
```

2.113 DIMENSION Statement

Description

The DIMENSION statement specifies the shape of an array.

Syntax

```
DIMENSION [ :: ] array-name ( array-spec ) [ , array-name ( array-spec ) ] ...
```

Where:

array-name is the name of an array.

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
```

explicit-shape-spec-list is a comma-separated list of

```
[ lower-bound : ] upper-bound
```

that specifies the shape and bounds of an explicit-shape array.

lower-bound is a specification expression (that can be evaluated on entry to the program unit) that specifies the lower bound of a given dimension of the array. If the *lower-bound* is omitted, the default value is 1.

upper-bound is a specification expression (that can be evaluated on entry to the program unit) that specifies the upper bound of a given dimension of the array.

If the *upper-bound* is less than the *lower-bound* the range is empty, the extent in that dimension is zero, and the array is of zero size.

explicit-shape-spec-list specifies explicit values for the bounds in each dimension of the array. An array that is declared with an *explicit-shape-spec-list* is an explicit-shape array. An explicit-shape array whose bounds depend on the values of nonconstant expressions shall be a dummy argument, a function result, or an automatic array of a procedure.

assumed-shape-spec-list is a comma-separated list of

```
[ lower-bound ] :
```

An array that is declared with an a*ssumed-shape-spec-list* is an assumed-shape array. An assumed shape array is a nonpointer dummy argument array that takes its shape from the associated actual argument array.

deferred-shape-spec-list is a comma-separated list of

```
:
```

that specifies the rank of a deferred-shape array. A deferred-shape array is an allocatable array or an array pointer.

assumed-size-spec is

```
[ explicit-shape-spec-list , ] [ lower-bound : ] *
```

assumed-size-spec specifies the shape of a dummy argument array whose size is assumed from the corresponding actual argument array that is called an assumed-size array.

An array that is declared with an *implied-shape-spec-list* is an implied shape array. See "1.5.15 Implied Shape Array" for implied shape array.

Example

```
subroutine sub(e,f,i)
dimension a(3,2,1) ! a is a 3x2x1 array
dimension b(-3:3)  ! b is a 7-element vector with a
                   ! lower bound of -3
dimension c(1:i)   ! c is an automatic array with a
                   ! upper bound of i
dimension d(:,:,:) ! d is deferred-shape array of
                   ! rank 3
dimension e(:,:)   ! e is an assumed-shape array of rank 2
dimension f(*)     ! f is an assumed-size array
allocatable :: d
```

2.114 DO Construct

Description

The DO construct specifies the repeated execution (loop) of a sequence of statements or executable constructs.

Syntax

```
[ do-construct-name : ] DO [ label ] [ loop-control ]
        block
do-term-stmt
```

Where:

do-construct-name is an optional name given to the DO construct.

label is the optional label of a statement that terminates the DO construct.

loop-control is

```
[ , ] do-variable=scalar-expr,scalar-expr[,scalar-expr]        or
[ , ] WHILE ( scalar-logical-expr )                            or
[ , ] UNTIL ( scalar-logical-expr )                            or
[ , ] CONCURRENT forall-header
```

do-variable is a named scalar variable of type INTEGER, default REAL, or double precision REAL. The *do-variable* of type default REAL or double precision REAL is deleted feature.

scalar-expr is a scalar expression of type INTEGER, default REAL, or double precision REAL. The *scalar-expr* is of type default REAL or double precision REAL is deleted feature.

scalar-logical-expr is a scalar LOGICAL expression.

See "1.16 FORALL Header" for *forall-header*.

block is a sequence of zero or more statements or executable constructs. *block* need not contain any executable statements and constructs. Execution of such a *block* has no effect.

do-term-stmt is

```
END DO[ do-construct-name ]     or
CONTINUE                        or
action-stmt
```

action-stmt statement is an action statement other than a CONTINUE, GO TO, RETURN, STOP, EXIT, CYCLE, END, or arithmetic IF statement.

If the DO statement is identified by a *do-construct-name* and the corresponding *do-term-stmt* is the END DO statement, the END DO statement shall specify the same *do-construct-name*. If the DO statement is not identified by a *do-construct-name*, the corresponding *do-term-stmt* shall not specify a *do-construct-name*.

If optional *label* is specified in the DO statement, the corresponding *do-term-stmt* shall be identified with the same *label*. Otherwise, the corresponding *do-term-stmt* shall be the END DO statement.

Remarks

A block DO construct is a DO construct that has an END DO or a CONTINUE *do-term-stmt* and it's not shared with other DO construct. A nonblock DO construct is a DO construct that has an *action-stmt* for *do-term-stmt* or has a *do-term-stmt* that is shared with other DO construct.

When the DO statement is executed, the DO construct becomes active.

If *loop-control* has *do-variable*, the initial parameter m_1, the terminal parameter m_2, and the incrementation parameter m_3 are established by evaluating each *scalar-expr* as the type of *do-variable*. If the third *scalar-expr* does not appear, m_3 has the value 1. The DO variable becomes defined with the value of the initial parameter m_1. The iteration count is established and is the value of the expression MAX((m_2-m_1+m_3)/m_3,0) when *do-variable* is of type INTEGER, and is the value of the expression MAX(INT((m_2-m_1+m_3)/m_3),0) when *do-variable* is of type default REAL or double precision REAL, and the range of DO loop is executed iteration count times. An iteration count of zero is possible. The DO variable is incremented by the value of the incrementation parameter m_3 for each range of DO loop execution. After execution of the range of DO loop iteration count times, the DO construct becomes inactive.

If there is no *loop-control* it is as if the iteration count were effectively infinity.

If *loop-control* is [,] WHILE (*scalar-logical-expr*), the effect is as if *loop-control* was omitted and the following statement inserted as the first statement of the *block*:

```
IF (.NOT. ( scalar-logical-expr ) ) EXIT
```

If *loop-control* is [,] UNTIL (*scalar-logical-expr*), the effect is as if *loop-control* was omitted and the following statement inserted as the last statement of the *block*:

```
IF ( scalar-logical-expr ) EXIT
```

The execution of the DO range may be curtailed by executing a CYCLE statement from within the range of the loop.

The EXIT statement provides one way of terminating a loop.

If *loop-control* is [,] CONCURRENT *forall-header*, the following specifications are applied:

- The execution order of the iterations is indeterminate.
- A CYCLE statement (see "2.99 CYCLE Statement") shall not appear if it belongs to an outer construct.
- A branch shall not have a branch target that is outside the construct.
- An EXIT statement (see "2.162 EXIT Statement") , RETURN statement (see "2.397 RETURN Statement") , or STOP statement (see "2.443 STOP Statement") shall not appear.
- Procedure IEEE_GET_FLAG (see "2.226 IEEE_GET_FLAG Intrinsic Module Subroutine") , IEEE_SET_HALTING_MODE (see "2.242 IEEE_SET_HALTING_MODE Intrinsic Module Subroutine") , or IEEE_GET_HALTING_MODE (see "2.227 IEEE_GET_HALTING_MODE Intrinsic Module Subroutine") for the intrinsic module IEEE_EXCEPTIONS (see "1.15.2 IEEE Exceptions") to reference shall not appear.
- An EXIT service subroutine (see "2.163 EXIT Service Subroutine") to reference shall not appear.
- A reference to procedure shall be a pure procedure.
- A variable that is referenced in an iteration either shall be previously defined during that iteration, or shall not be defined during any other iteration.
- A variable that is defined by more than one iteration becomes undefined when the loop terminates.
- A pointer that is referenced in an iteration either shall be previously pointer associated during that iteration, or shall not have its pointer association changed during any iteration.

- A pointer that has its pointer association changed in more than one iteration has an association status of undefined when the loop terminates.
- An allocatable object that is allocated in more than one iteration shall be deallocated during the same iteration in which it was allocated. An object that is allocated or deallocated in only one iteration shall not be deallocated, allocated, referenced, or defined in a different iteration.
- An input/output statement shall not output in one iteration and input from the same record in a different iteration.

Example

```
! loop-control of DO construct has do-variable
do i=1,100          ! iterates 100 times
  do while (a>b)    ! iterates while a>b
    do 10 j=1,100,3 ! iterates 34 times
      ...
10  continue
  end do
end do
```

The CYCLE statement can be used to curtail execution of the current iteration of a DO loop. The EXIT statement can be used to exit a DO loop altogether.

```
! loop-control of DO construct has CONCURRENT
real,dimension(5)::a,b,c
...
do concurrent(k=1:5)
  c(k) = a(k)+sqrt(b(k))
end do
```

2.115 DO Statement

Description

The DO statement begins a DO construct. See "2.114 DO Construct" for DO construct.

Syntax

```
[ do-construct-name : ] DO [ label ] [ loop-control ]
```

Where:

do-construct-name is an optional name given to the DO construct.

label is the optional label of a statement that terminates the DO construct.

loop-control is

```
[ , ] do-variable = scalar-expr , scalar-expr [ , scalar-expr ]        or
[ , ] WHILE ( scalar-logical-expr )                                    or
[ , ] UNTIL ( scalar-logical-expr )                                    or
[ , ] CONCURRENT  forall-header
```

do-variable is a named scalar variable of type INTEGER, default REAL, or double precision REAL. The *do-variable* of type default REAL or double precision REAL is deleted feature.

scalar-expr is a scalar expression of type INTEGER, default REAL, or double precision REAL. The *scalar-expr* is of type default REAL or double precision REAL is deleted feature.

scalar-logical-expr is a scalar LOGICAL expression.

See "1.16 FORALL Header" for *forall-header*.

See "2.114 DO Construct" for DO construct.

2.116 DOT_PRODUCT Intrinsic Function

Description

Dot-product multiplication of vectors.

Class

Transformational function.

Syntax

```
result = DOT_PRODUCT ( VECTOR_A , VECTOR_B )
```

Argument(s)

VECTOR_A shall be of type INTEGER, REAL, COMPLEX, or LOGICAL. It shall be array-valued and of rank one.

VECTOR_B shall be of numeric type if *VECTOR_A* is of numeric type and of type LOGICAL if *VECTOR_A* is of type LOGICAL. It shall be array-valued, of rank one, and of the same size as *VECTOR_A*.

Result

If the arguments are of type LOGICAL, then the result is scalar and of type default LOGICAL. Its value is ANY (*VECTOR_A* .AND. *VECTOR_B*). If the vectors have size zero, the result has the value false.

If the arguments are of different numeric type, then the result type is that of the argument with the higher type, where COMPLEX is higher than REAL, and REAL is higher than INTEGER. If both arguments are of the same type, the result kind is the kind of the argument that offers the greater range.
The result value is SUM (*VECTOR_ A* * *VECTOR_B*) if *VECTOR_A* is of type REAL or INTEGER.
The result value is SUM (CONJG (*VECTOR_A*) * *VECTOR_B*) if *VECTOR_A* is of type COMPLEX.

Example

```
i = dot_product((/3,4,5/),(/6,7,8/))
        ! i is assigned the value 86
```

2.117 DOUBLE PRECISION Type Declaration Statement

Description

The DOUBLE PRECISION type declaration statement declares entities of type double precision REAL. See "2.471 TYPE Type Declaration Statement" for type declaration statement.

Syntax

```
DOUBLE PRECISION [ [ , attr-spec ] ... :: ] entity-decl-list
```

2.118 DPROD Intrinsic Function

Description

Double-precision REAL product.

Class

Elemental function.

Syntax

```
result = DPROD ( X , Y )
```

Argument(s)

X

X shall be of type default REAL.

Y

Y shall be of type default REAL.

Result

The result is of type double-precision REAL. Its value is a double precision representation of the product of *X* and *Y*.

Example

```
double precision :: dub
dub = dprod (3.e2, 4.4e4) ! dub is assigned 13.2d6
```

2.119 DRAND Service Function

Description

Generates a random number greater than or equal to 0.0 and less than or equal to 1.0.

Syntax

```
dy = DRAND ( i )
```

Argument(s)

i

Default INTEGER scalar.

Result

REAL(8), scalar. The values are as follows:

Argument(*i*)	Selection process
0	The next random number in the sequence is selected.
1	The generator is restarted and the first random value is selected.
Otherwise	The generator is reseeded using *i*, restarted, and the first random value is selected.

Example

```
use service_routines,only:drand
do i=1,10
  print *,drand(0)   ! Generates 10 random numbers.
end do
end
```

2.120 DSHIFTL Intrinsic Function

Description

Combined left shift.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
DSHIFTL	---	3	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal	INTEGER

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
			constant , INTEGER	

```
result = DSHIFTL ( I , J , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER, or a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER, or a binary, octal or hexadecimal constant.

SHIFT

SHIFT shall be of type INTEGER. The value shall be less than or equal to BIT_SIZE(*I*) or BIT_SIZE(*J*), and shall be nonnegative.

Result

The type of the result is the same as the type INTEGER of the *I* or *J*.

Remarks

I and *J* are combined, and it shift to left. The result is same as

IOR (SHIFTL (*I*, *SHIFT*), SHIFTR (*J*, BIT_SIZE(*J*)-*SHIFT*)).

If *I* and *J* are of type INTEGER, *I* and *J* shall be the same as the type.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. The binary, octal or hexadecimal constants for both shall not specify.

The type of the result is the same as the type INTEGER of the *I* or *J*.

Example

```
k = dshiftl(1, X'40000000', 2)    ! k is assigned the value 5
```

2.121 DSHIFTR Intrinsic Function

Description

Combined right shift.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
DSHIFTR	---	3	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant , INTEGER	INTEGER

```
result = DSHIFTR ( I , J , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER, or a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER, or a binary, octal or hexadecimal constant.

SHIFT

SHIFT shall be of type INTEGER. The value shall be less than or equal to BIT_SIZE(*I*) or BIT_SIZE(*J*), and shall be nonnegative.

Result

The type of the result is the same as the type INTEGER of the *I* or *J*.

Remarks

I and *J* are combined, and it shift to right. The result is same as

IOR (SHIFTL (*I*, BIT_SIZE(*I*)-*SHIFT*), SHIFTR (*J*, *SHIFT*)).

If *I* and *J* are of type INTEGER, *I* and *J* shall be the same as the type. If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. The binary, octal or hexadecimal constants for both shall not specify.

The type of the result is the same as the type INTEGER of the *I* or *J*.

Example

```
k = dshiftr(1, 16, 3)    ! k is assigned the value 536870914
```

2.122 DTIME Service Function

Description

Returns the CPU time since the start of program execution when first called, and the CPU time since the last call to DTIME thereafter.

Syntax

```
y = DTIME ( tm )
```

Argument(s)

tm

Default REAL and rank one. Its size shall be at least 2. The values returned in *tm* are as follows:

```
tm(1) : Elapsed user time
tm(2) : Elapsed system time
```

Result

Default REAL scalar. Total CPU time in seconds, which is the sum of *tm*(1) and *tm*(2).

On the first call, CPU time since the start of execution. On second and subsequent calls, CPU time since last call to DTIME.

If there is an error, argument elements *tm*(1) and *tm*(2) are undefined and function return value = -1.0.

Example

```
use service_routines,only:dtime
real :: tm(2),y
tm = 1.0
y=dtime(tm)
do i=1,5000
  write(*,*) i,i*i
end do
y=dtime(tm)
end
```

2.123 DVCHK Service Subroutine

Description

Tests for a division by zero.

Syntax

```
CALL DVCHK ( i )
```

Argument(s)

i

Default INTEGER scalar. The DVCHK service subroutine tests for a divide-check exception and returns a value that indicates the existing condition. After testing, the divide check indicator is turned off. The value of *i* is returned by the service subroutine to indicate the following:

```
=1:Division by zero.
=2:No division by zero.
```

Example

```
use service_routines,only:dvchk
integer :: i0, i1, i2
real :: f = 0.0
call dvchk(i0)  ! i0 .eq. 2
call sub(f)
call dvchk(i1)  ! i1 .eq. 1
call dvchk(i2)  ! i2 .eq. 2
end
subroutine sub(zero)
real, intent(inout) :: zero
zero = 1.0 / zero   ! divide by zero
end
```

2.124 ELSE Statement

Description

The ELSE statement controls conditional execution of a following block in an IF construct where all previous IF expressions are false. See "2.263 IF Construct" for IF construct.

Syntax

```
ELSE [ if-construct-name ]
```

Where:

if-construct-name is the optional name given to the IF construct.

2.125 ELSE IF Statement

Description

The ELSE IF statement specifies the condition to execute the following block in an IF construct where all previous IF expressions are false. See "2.263 IF Construct" for IF construct.

Syntax

```
ELSE IF ( scalar-logical-expr ) THEN [ if-construct-name ]
```

Where:

scalar-logical-expr is a scalar LOGICAL expression.

if-construct-name is the optional name given to the IF construct.

2.126 ELSEWHERE Statement

Description

The ELSEWHERE statement controls conditional execution of a block of assignment statements for elements of an array for which the WHERE construct's mask expression is false. See "2.486 WHERE Construct" for WHERE construct.

Syntax

```
ELSEWHERE ( mask-expr ) [ where-construct-name ]        or
ELSEWHERE [ where-construct-name ]
```

Where:

mask-expr is an array LOGICAL expression. An ELSEWHERE statement that has a *mask-expr* is a masked ELSEWHERE statement.

where-construct-name is the optional name given to the WHERE construct.

2.127 END Statement

Description

An END PROGRAM statement, END FUNCTION statement, END SUBROUTINE statement, END MODULE statement, or END BLOCK DATA statement is an END statement.

The END PROGRAM, END FUNCTION, and END SUBROUTINE statements are executable and can be branch target statements. Executing an END PROGRAM statement terminates the executing program. Executing an END FUNCTION or END SUBROUTINE statement is equivalent to executing a RETURN statement in a subprogram.

The END MODULE and END BLOCK DATA statements are non-executable.

See "2.141 END PROGRAM Statement", "2.136 END FUNCTION Statement", "2.144 END SUBROUTINE Statement","2.140 END MODULE Statement", or "2.130 END BLOCK DATA Statement" for each statement.

2.128 END ASSOCIATE Statement

Description

The END ASSOCIATE statement ends an ASSOCIATE construct. See "2.28 ASSOCIATE Construct" for ASSOCIATE construct.

Syntax

```
END ASSOCIATE [ associate-construct-name ]
```

Where:

associate-construct-name is the name of ASSOCIATE construct.

2.129 END BLOCK Statement

Description

The END BLOCK statement ends a BLOCK construct. See "2.50 BLOCK Construct" for BLOCK construct.

Syntax

```
END BLOCK [ block-construct-name ]
```

Where:

block-construct-name is the name of BLOCK construct.

2.130 END BLOCK DATA Statement

Description

The END BLOCK DATA statement ends a block data program unit. The END BLOCK DATA statement is a nonexecutable statement. See "1.11.3 Block Data Program Units" for block data program unit.

Syntax

```
END [ BLOCK DATA [ block-data-name ] ]
```

Where:

block-data-name is the name of the block data program unit.

If a *block-data-name* is present in the END BLOCK DATA statement, it shall be identical to the *block-data-name* specified in the BLOCK DATA statement.

Example

```
block data blkd
  common /com/ a,b,c
  integer :: a=1,b=2,c=3
end block data blkd
```

2.131 END CRITICAL Statement

Description

The END CRITICAL statement ends a CRITICAL construct. See "2.96 CRITICAL Construct" for CRITICAL construct.

Syntax

```
END CRITICAL [ critical-construct-name ]
```

Where:

critical-construct-name is the name of CRITICAL construct.

2.132 END DO Statement

Description

The END DO statement ends a DO construct. See "2.114 DO Construct" for DO construct.

Syntax

```
END DO [ do-construct-name ]
```

Where:

do-construct-name is the name of the DO construct.

2.133 END ENUM Statement

Description

The END ENUM statement specifies the end of an enumeration body definition. See "1.5.11.7 Enumeration Bodies and Enumerators", for information on enumeration body definitions.

Syntax

```
END ENUM
```

2.134 ENDFILE Statement

Description

The ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record, which becomes the last record of the file.

Execution of the ENDFILE statement performs a wait operation on all unfinished asynchronous data transfer operations at the specified device.

Syntax

```
ENDFILE external-file-unit          or
ENDFILE ( position-spec-list )
```

Where:

external-file-unit is a scalar INTEGER expression corresponding to the input/output unit number of an external file.

position-spec-list is a comma-separated list of

```
[ UNIT = ] external-file-unit        or
IOMSG = iomsg                        or
IOSTAT = io-stat                     or
ERR = err-label
```

position-spec-list shall contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

If the optional characters UNIT= are omitted from the *external-file-unit* specifier, the *external-file-unit* specifier shall be the first item in the *position-spec-list*.

iomsg must be a scalar default CHARACTER variable. If the error conditions occur during input/output statement execution, an explanatory message is assigned to *iomsg*.

io-stat must be a variable of type INTEGER that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, and zero otherwise.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the ENDFILE statement. If an error condition occurs during execution of the ENDFILE statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

Remarks

After execution of an ENDFILE statement, a BACKSPACE (see "2.41 BACKSPACE Statement") or REWIND statement (see "2.398 REWIND Statement") shall be executed to reposition the file before any data transfer statement or subsequent ENDFILE statement.

When the ENDFILE statement is executed in a file connected as a stream access, the current file position becomes the file end point. It is assumed that only file storage units preceding the current position have already been written, and subsequently only these file storage units can be read. Subsequent stream output statements can write additional data to the file.

An ENDFILE statement on a file that is connected but does not yet exist causes the file to be created before writing the endfile record.

Files connected as a direct access cannot be referenced by an ENDFILE statement. Files in which the ACTION= specifier value equals 'READ', that cannot be referenced by the ENDFILE statement.

Example

```
endfile 10  ! writes an endfile record to the file
            ! connected to unit 10
```

2.135 END FORALL Statement

Description

The END FORALL statement ends a FORALL construct. See "2.176 FORALL Construct" for FORALL construct.

Syntax

```
END FORALL [ forall-construct-name ]
```

Where:

forall-construct-name is the name of the FORALL construct.

2.136 END FUNCTION Statement

Description

The END FUNCTION statement ends a function subprogram. The END FUNCTION statement is executable and can be branch target statement. Executing an END FUNCTION statement is equivalent to executing a RETURN statement in a function subprogram. See "1.12.1 Function Subprogram" for function subprogram.

Syntax

```
END [ FUNCTION [ function-name ] ]
```

Where:

FUNCTION shall be present in the END FUNCTION statement of an internal or module function.

function-name is the name of the function subprogram.

If a *function-name* is present in the END FUNCTION statement, it shall be identical to the *function-name* specified in the FUNCTION statement.

Example

```
function foo(i,j)
  integer :: foo,i,j
  foo = i*(j+2)
end function foo
```

2.137 END IF Statement

Description

The END IF statement ends an IF construct. See "2.263 IF Construct" for IF construct.

Syntax

```
END IF [ if-construct-name ]
```

Where:

if-construct-name is the name of the IF construct.

2.138 END INTERFACE Statement

Description

The END INTERFACE statement ends an interface block. See "1.12.7.2 Procedure Interface Block" for interface block.

Syntax

```
END INTERFACE [ generic-spec ]
```

Where:

generic-spec is

```
generic-name                      or
OPERATOR ( defined-operator )     or
ASSIGNMENT (=)                    or
dtio-generic-spec
```

If the *generic-spec* is specified, it shall be identical to the *generic-spec* specified in INTERFACE statement.

generic-name is the name of the generic procedure.

defined-operator is

```
intrinsic-operator        or
. operator-name .
```

intrinsic-operator is an intrinsic operator.

operator-name is a user-defined name for the operation, consisting of one to 240 letters.

ASSIGNMENT(=) is a user-defined assignment.

dtio-generic-spec is derived type input/output procedure that is the following syntax:

```
READ ( FORMATTED )          or
READ ( UNFORMATTED )        or
WRITE ( FORMATTED )         or
WRITE ( UNFORMATTED )
```

Example

```
interface                           ! without a generic-spec
  subroutine ex(a,b,c)
    implicit none
    real ,dimension(2,8) :: a,b,c
    intent(in) :: a
    intent(out) :: b
  end subroutine ex
  function why(t,f)
    implicit none
    logical ,intent(in) :: t,f
    logical :: why
  end function why
end interface
interface swap                      ! swap is a generic-name
  subroutine real_swap(x,y)
    implicit none
    real ,intent(inout) :: x,y
  end subroutine real_swap
  subroutine int_swap(x,y)
    implicit none
    integer ,intent(inout) :: x,y
  end subroutine int_swap
end interface swap
interface operator(*)               ! defined operator *
  function set_intersection(a,b)
    use set_module
    implicit none
    type(set) , intent(in) :: a,b
    type(set) :: set_intersection
  end function set_intersection
end interface operator(*)
interface assignment(=)             ! defined assignment
  subroutine integer_to_bit(n,b)
    implicit none
    logical ,intent(out) :: n
    integer ,intent(in) :: b
  end subroutine integer_to_bit
end interface assignment(=)
```

2.139 END MAP Statement

Description

The END MAP statement ends a block of union declaration in derived type definition by STRUCTURE statement. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
END MAP
```

Example

```
structure /complex_element/
  union
    map
      real :: real,imag
    end map
    map
      complex :: complex
    end map
  end union
end structure
record /complex_element/ x
x%real = 2.0
x%imag = 3.0
print *,x%complex     ! complex has the value (2.0,3.0)
```

2.140 END MODULE Statement

Description

The END MODULE statement ends a module. The END MODULE statement is a nonexecutable statement.
See "1.11.2 Modules" for module.

Syntax

```
END [ MODULE [ module-name ] ]
```

Where:

module-name is the name of the module.

If a *module-name* is present in the END MODULE statement, it shall be identical to the *module-name* specified in the MODULE statement.

Example

```
module mod
  implicit none
  type mytype
    real :: a,b(2,4)
    integer :: n,o,p
  end type mytype
end module mod
subroutine zee()
  use mod
  implicit none
  type (mytype) :: bee, dee
  ...
end subroutine zee
```

2.141 END PROGRAM Statement

Description

The END PROGRAM statement ends a main program. The END PROGRAM statement is executable and can be branch target statement. Executing an END PROGRAM statement terminates the executing program. See "1.11.1 Main Program" for main program.

Syntax

```
END [ PROGRAM [ program-name ] ]
```

Where:

program-name is the name of the program.

If a *program-name* is present in the END PROGRAM statement, it shall be identical to the *program-name* specified in the PROGRAM statement.

Example

```
program main
  ...
end program main
```

2.142 END SELECT Statement

Description

The END SELECT statement ends a CASE construct or a SELECT TYPE construct. See "2.56 CASE Construct" for CASE construct. See "2.413 SELECT TYPE Construct" for SELECT TYPE construct.

Syntax

```
END SELECT [ case-construct-name ]
```

Where:

case-construct-name is the name of the CASE construct or the SELECT TYPE construct.

2.143 END STRUCTURE Statement

Description

The END STRUCTURE statement ends a derived type definition by STRUCTURE statement. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
END STRUCTURE
```

Example

```
structure /complex_element/
  union
    map
      real :: real,imag
    end map
    map
      complex :: complex
    end map
  end union
end structure
record /complex_element/ x
x%real = 2.0
x%imag = 3.0
print *,x%complex     ! complex has the value (2.0,3.0)
```

2.144 END SUBROUTINE Statement

Description

The END SUBROUTINE statement ends a subroutine subprogram. The END SUBROUTINE statement is executable and can be branch target statement. Executing an END SUBROUTINE statement is equivalent to executing a RETURN statement in a subroutine subprogram. See "1.12.2 Subroutine Subprogram" for subroutine subprogram.

Syntax

```
END [ SUBROUTINE [ subroutine-name ] ]
```

Where:

SUBROUTINE shall be present in the END SUBROUTINE statement of an internal or module subroutine.

subroutine-name is the name of the subroutine subprogram.

If a *subroutine-name* is present in the END SUBROUTINE statement, it shall be identical to the *subroutine-name* specified in the SUBROUTINE statement.

Example

```
pure subroutine zee (var1,var2)
  ...
end subroutine zee
```

2.145 END TYPE Statement

Description

The END TYPE statement ends a derived type definition by TYPE statement. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
END TYPE [ type-name ]
```

Where:

type-name is the name of the derived type.

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

2.146 END UNION Statement

Description

The END UNION statement ends a union declaration in derived type definition by STRUCTURE statement. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
END UNION
```

Example

```
structure /complex_element/
  union
    map
      real :: real,imag
    end map
    map
      complex :: complex
    end map
  end union
end structure
record /complex_element/ x
x%real = 2.0
x%imag = 3.0
print *,x%complex     ! complex has the value (2.0,3.0)
```

2.147 END WHERE Statement

Description

The END WHERE statement ends a WHERE construct. See "2.486 WHERE Construct" for WHERE construct.

Syntax

```
END WHERE [ where-construct-name ]
```

Where:

where-construct-name is the name of the WHERE construct.

2.148 ENTRY Statement

Description

The ENTRY statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which ENTRY statement appears. An ENTRY statement may appear only in an external subprogram or module subprogram.

Syntax

```
ENTRY entry-name [ ( [ dummy-arg-list ] ) [ suffix ] ]
```

Where:

entry-name is the name of the entry.

dummy-arg-list is a comma-separated list of

```
dummy-arg-name        or
*
```

dummy-arg-name is the name of the dummy argument.

A *dummy-arg* may be an * only if the ENTRY statement is in a subroutine subprogram.

suffix is

```
proc-language-binding-spec [ RESULT ( result-name ) ]        or
RESULT ( result-name ) proc-language-binding-spec
```

result-name is the name of the result.

Remarks

An ENTRY statement shall not appear within an executable construct.

RESULT may be present only if the ENTRY statement is in a function subprogram. If RESULT is specified, the *entry-name* shall not appear in any specification or type declaration statement in the scoping unit of the function subprogram. If RESULT is specified, *result-name* shall not be the same as *entry-name*.

If the ENTRY statement is in a function subprogram, an additional function is defined by that subprogram. The name of the function is *entry-name* and its result variable is *result-name* or is *entry-name* if no *result-name* is provided. The characteristics of the function result are specified by specifications of the result variable. If the characteristics of the result of the function named in the ENTRY statement are the same as the characteristics of the result of the function named in the FUNCTION statement, their result variables identify the same variable. Otherwise, they are storage associated and shall all be scalars without the POINTER attribute.

If the ENTRY statement is in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is *entry-name*.

Example

```
program main
  i=2
  call square(i)
  j=2
```

```
  call quad(j)
  print*, i,j   ! prints 4         16
end program main
subroutine quad(k)
  k=k*k
entry square(k)
  k=k*k
  return
end subroutine quad
```

2.149 ENUM Statement

Description

The ENUM statement specifies the start of an enumeration body definition.
See "1.5.11.7 Enumeration Bodies and Enumerators", for information on enumeration body definitions.

Syntax

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

2.150 ENUMERATOR Statement

Description

The ENUMERATOR statement defines an enumerator, which is an enumeration body definition entity.
See "1.5.11.7 Enumeration Bodies and Enumerators", for information on enumeration body definitions.

Syntax

```
ENUMERATOR [::] named-constant [ = scalar-int-initialization-expr ]
```

Where:

named-constant is a list of default INTEGER type of named constant of enumerators.

scalar-int-initialization-expr is a scalar INTEGER initialization expression.

If = is written in the enumerator, two consecutive colons must be written as delimiters before the enumerator list.

2.151 EOSHIFT Intrinsic Function

Description

End-off shift of all rank one sections in an array. Elements are shifted out at one end and copies of boundary values are shifted in at the other. Different sections can be shifted by different amounts and in different directions by using an array-valued shift.

Class

Transformational function.

Syntax

```
result = EOSHIFT ( ARRAY , SHIFT [ , BOUNDARY , DIM ] )
```

Required Argument(s)

ARRAY

ARRAY can be of any type. It shall not be scalar.

SHIFT

SHIFT shall be of type INTEGER and shall be scalar if *ARRAY* is of rank one; otherwise it shall be scalar or of rank n-1 and of shape $(d_1,d_2,...,d_{DIM-1},d_{DIM+1},...,d_n)$, where $(d_1,d_2,...,d_n)$ is the shape of *ARRAY*.

Optional Argument(s)

BOUNDARY

BOUNDARY shall be of the same type and kind as *ARRAY*. If *ARRAY* is of type CHARACTER, *BOUNDARY* shall have the same length as *ARRAY*. It shall be scalar if *ARRAY* is of rank one; otherwise it shall be scalar or of rank *n*-1 and of shape (d_1,d_2,...,d_{DIM-1},d_{DIM+1},...,d_n) . *BOUNDARY* can be omitted, in which case the default values are zero for numeric types, blanks for CHARACTER, and false for LOGICAL.

DIM

DIM shall be a scalar INTEGER with a value in the range 1 <= *DIM* <= *n* , where *n* is the rank of *ARRAY*. If *DIM* is omitted, it is as if it were present with a value of one.

Result

The result is of the same type, kind and shape as *ARRAY*.

Element(s_1,s_2,...,s_n) of the result has the value *ARRAY*(s_1,s_2,...,s_{DIM-1},s_{DIM+sh},s_{DIM+1},...,s_n) where *sh* is *SHIFT* or *SHIFT*(s_1,s_2,...,s_{DIM-1},s_{DIM+1},...,s_n) provided the inequality LBOUND(*ARRAY*,*DIM*)<=s_{DIM+sh}<=UBOUND(*ARRAY*,*DIM*) holds and is otherwise *BOUNDARY* or *BOUNDARY*(s_1,s_2,...,s_{DIM-1},s_{DIM+1},...,s_n).

Example

```
integer, dimension (2,3) :: a, b
integer, dimension (3) :: c, d
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                  !  represents  |1 3 5|
                  !              |2 4 6|
c = (/1,2,3/)
b = eoshift(a,1)   ! b is assigned the value |2 4 6|
                  !                         |0 0 0|
b = eoshift(a,-1,0,2) ! b assigned the value |0 1 3|
                  !                         |0 2 4|
b = eoshift(a,-c,1)! b is assigned the value |1 1 1|
                  !                         |1 1 1|
d = eoshift(c,2)   ! d is assigned the value |3 0 0|
```

2.152 EPSILON Intrinsic Function

Description

Positive value that is almost negligible compared to unity; smallest *x* such that 1+*x* is not equal to 1.

Class

Inquiry function.

Syntax

result = EPSILON (*X*)

Argument(s)

X

X shall be of type REAL. It can be scalar or array-valued.

Result

The result is a scalar value of the same kind as *X*. Its value is 2^{1-p}, where *p* is the number of bits in the fraction part of the physical representation of *X*.

The result value is as follows:

Type of *X*	The result value
Single precision REAL	1.19209290E-07

Type of *X*	The result value
Double precision REAL	2.220446049250313D-16
Quadruple precision REAL	1.92592994438723585305597794258492 73Q-0034

Example

```
! reasonably safe compare of two default REALs
function equals (a, b)
  implicit none
  logical :: equals
  real, intent(in) :: a, b
  real :: eps
  eps = abs(a) * epsilon(a)  ! scale epsilon
  if (eps == 0) then
    eps = tiny (a)        ! if eps underflowed to 0
                          ! use a very small
                          ! positive value for epsilon
  end if
  if (abs(a-b) > eps) then
    equals = .false.     ! not equal if difference>eps
  else
    equals = .true.      ! equal otherwise
  end if
  return
end function equals
```

2.153 EQUIVALENCE Statement

Description

The EQUIVALENCE statement is used to specify that two or more objects in a scoping unit share the same storage; called storage association. If the equivalenced objects have differing type or type parameters, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a scalar and an array are equivalenced, the scalar does not have array properties and the array does not have the properties of a scalar.

Syntax

`EQUIVALENCE` *`equivalence-set-list`*

Where:

equivalence-set-list is a comma-separated list of

`(` *`equivalence-object`* `,` *`equivalence-object-list`* `)`

equivalence-object-list is a comma-separated list of *equivalence-object*.

equivalence-object is

`variable-name` or
`array-element` or
`substring`

variable-name is the name of the variable.

array-element is the array element whose each subscript expression shall be an INTEGER initialization expression.

substring is the substring whose each substring range expression shall be an INTEGER initialization expression.

Remarks

An *equivalence-object* shall not be a dummy argument, a pointer, an allocatable variable, an object of a derived type containing an ultimate component that is an allocatable variable, an object of a non-sequence derived type, an object of a sequence derived type containing a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a variable with BIND attribute, a named constant, a structure component, or a subobject of any of the preceding objects.

An *equivalence-object* shall not have the TARGET attribute. The name of an *equivalence-object* shall not be a name made accessible by use association.

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.

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. The length of the equivalenced objects need not be the same.

If an *equivalence-object* is of a 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.

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.

An EQUIVALENCE statement shall not specify that the same storage unit is to occur more than once in a storage sequence. An EQUIVALENCE statement shall not specify that consecutive storage units are to be nonconsecutive.

Example

```
equivalence (a,b,c(2)) ! a, b, and c(2) share the
                       ! same storage
```

2.154 ERF Intrinsic Function

Description

Error function.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ERF	ERF	1	Single-prec REAL	Single-prec REAL
	DERF		Double-prec REAL	Double-prec REAL
	QERF		Quad-prec REAL	Quad-prec REAL
ERFC	ERFC	1	Single-prec REAL	Single-prec REAL
	DERFC		Double-prec REAL	Double-prec REAL
	QERFC		Quad-prec REAL	Quad-prec REAL

```
result = ERF ( X )
result = ERFC ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type as *X*. Its value is the error function for data of REAL type.

Remarks

ERF, DERF, and QERF evaluate the error function for data of REAL type. The range is -1 <= ERF(*X*) <= 1 .

ERFC, DERFC, and QERFC evaluate the complement of the error function for data of real type for very large values of *X*. The range is 0 <= ERFC(*X*) <= 2.

The generic names, ERF and ERFC, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = erf(.5)
```

2.155 ERROR STOP Statement

Description

The ERROR STOP statement terminates execution of the program as an error.

Syntax

```
ERROR STOP [ stop-code ]
```

Where:

stop-code is

scalar-default-char-initialization-expr or
scalar-int-initialization-expr

scalar-default-char-initialization-expr is scalar default CHARACTER initialization expression.

scalar-int-initialization-expr is scalar INTEGER initialization expression.

Remarks

When an ERROR STOP statement is executed, the diagnostic message is written to the standard error file and the executing program is terminated as an error.

Example

```
if (a > b) then
  error stop ! program execution terminated as an error
end if
```

2.156 ERROR Service Subroutine

Description

Sends a message to the standard output file.

Syntax

```
CALL  ERROR ( string )
```

Argument(s)

string

Default CHARACTER scalar. Message to go to the standard output file.

Example

```
   use service_routines,only:error
   open(10,file='x.dat',err=10)
   write(10,*)' Fortran program'
   stop
10 call error('Fortran I/O error')
   stop 200
   end
```

2.157 ERRSAV Service Subroutine

Description

ERRSAV saves the beginning of the error item (8 bytes) corresponding to *errno* in *darea*.

Syntax

```
CALL  ERRSAV ( errno , darea )
```

Argument(s)

errno

Default INTEGER scalar. Error number.

darea

Default CHARACTER scalar that length is 16 bytes. Variable in which the error item is saved.

If the type of *darea* is not character, this result is undefined.

Example

```
      use service_routines,only:errsav,errstr
      character(len=16) :: err113
!     (1)estop,(2)mprint,(3)ecount,(4)inf
      call errsav(113,err113)
      write(err113(1:2),'(2a1)') 0,0
      call errstr(113,err113)
      open(10,file='x.dat',form='formatted')
      do i=1,15
        write(10)i
      end do
      close(10,status='delete')
      end
```

2.158 ERRSET Service Subroutine

Description

The ERRSET service subroutine changes the contents of the control table corresponding to the first argument *errno*, storing the values specified in the second through sixth arguments.

Syntax

```
CALL  ERRSET ( errno , estop , mprint , trace , uexit , r )
```

Argument(s)

errno

Default INTEGER scalar. Error number.

estop

Default INTEGER scalar. Error count limit.

```
<=0      :     Error count limit is the same as before.
>=256    :     Indicates unlimited error counts.
```

mprint

Default INTEGER scalar. Message print limit.

```
<0       :     Message print limit is zero (that is, a no-print message).
=0       :     Message print limit is the same as before.
>=256    :     Indicates unlimited message prints.
```

trace

Default INTEGER scalar. One of the following values indicates whether to print a trace back map:

```
=0       :     Do not change.
=1       :     Do not print.
=2       :     Print.
```

uexit

Default INTEGER scalar. Error correction or user defined correction subroutine indicated with one of the following:

```
=0       :     Do not change.
=1       :     Perform standard correction.
=other   :     Execute the user defined correction.
```

r

Default INTEGER scalar. If *errno* is not 132, it indicates the largest error number to be changed.

If *errno* is 132, indicates whether a blank character is inserted at the beginning of correction data with the following values:

```
=1       :     Insert a blank character.
=other   :     Do not insert a blank character.
```

Example

```
use service_routines,only:errset
call errset(113,256,-1,1,0,0)
open(10,file='x.dat',form='formatted')
do i=1,15
  write(10) i
end do
close(10,status='delete')
end
```

2.159 ERRSTR Service Subroutine

Description

ERRSTR stores the error item of *darea* in the error item of the error control table corresponding to *errno*.

Syntax

```
CALL  ERRSTR ( errno , darea )
```

Argument(s)

errno

Default INTEGER scalar. Error number.

darea

Default CHARACTER scalar that length is 16 bytes. Error item to set.

Example

```
    use service_routines,only:errstr,errsav
    character(len=16) :: err113
!   (1)estop,(2)mprint,(3)ecount,(4)inf
    call errsav(113,err113)
    write(err113(1:2),'(2a1)')0,0
    call errstr(113,err113)
    open(10,file='x.dat',form='formatted')
    do i=1,15
      write(10) i
    end do
```

```
close(10,status='delete')
end
```

2.160 ERRTRA Service Subroutine

Description

The trace back map is generated for the program unit being executed.

Syntax

```
CALL  ERRTRA
```

Example

```
open(10,file='x.dat')
call sub1
end
subroutine sub1
use service_routines,only:errtra
call errtra
do i=1,15
  write(10,*) i
end do
end
```

2.161 ETIME Service Function

Description

Returns the elapsed CPU time since the start of program execution.

Syntax

```
y = ETIME ( tm )
```

Argument(s)

tm

Default REAL and rank one. Its size shall be at least 2. The values returned in *tm* are as follows:

```
tm(1) : Elapsed user time
tm(2) : Elapsed system time
```

Result

Default REAL scalar. Total CPU time in seconds, which is the sum of *tm*(1) and *tm*(2).

If there is an error, argument elements *tm*(1) and *tm*(2) are undefined and function return value is -1.0.

Example

```
 use service_routines,only:etime
 real :: tm1(2),tm2(2),y1,y2
 y1 = etime(tm1)
 do i=1,5000
   write(*,*) i,i*i
 end do
 y2 = etime(tm2)
 print *,'REAL TIME=',y2-y1,'USER TIME=',tm2(1)-tm1(1), &
&        'SYSTEM TIME=',tm2(2)-tm1(2)
 end
```

2.162 EXIT Statement

Description

The EXIT statement terminates an execution of DO loop.

Syntax

```
EXIT [ do-construct-name ]
```

Where:

do-construct-name is the name of a DO construct that contains the EXIT statement.

Execution of an EXIT statement causes to terminates the execution of the DO loop that identified by *do-construct-name*, if any. Otherwise, the execution of the DO loop in which the EXIT statement appears is terminated.

Example

```
outer: do i=1, 10
inner:   do j=1, 10
           if (i>a) exit outer
           if (j>b) exit ! exits inner
           ...
         end do inner
       end do outer
```

2.163 EXIT Service Subroutine

Description

The Fortran program is terminated.

Syntax

```
CALL  EXIT
```

Example

```
   use service_routines,only:exit,setrcd
   open(10,file='x.dat',err=10)
   do i=1,15
     write(10,*,err=10) i
   end do
   stop 'PROGRAM END'
10 call setrcd(130)
   call exit
   end
```

2.164 EXP Intrinsic Function

Description

Exponential.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
EXP	---	1	REAL or COMPLEX	REAL or COMPLEX
	EXP		Single-prec REAL	Single-prec REAL
	DEXP		Double-prec REAL	Double-prec REAL

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	QEXP		Quad-prec REAL	Quad-prec REAL
	CEXP		Single-prec COMPLEX	Single-prec COMPLEX
	CDEXP		Double-prec COMPLEX	Double-prec COMPLEX
	CQEXP		Quad-prec COMPLEX	Quad-prec COMPLEX

```
result = EXP ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type as *X*. Its value is a REAL or COMPLEX representation of e^x. If *X* is of type COMPLEX, its imaginary part is treated as a value in radians.

Remarks

EXP, DEXP, QEXP, CEXP, CDEXP, and CQEXP evaluate the natural exponential of REAL or COMPLEX data.

For a REAL(KIND=4) argument, the domain shall be *X* < 88.722E0 .

For a REAL(KIND=8) argument, the domain shall be *X* < 709.782D0 .

For a REAL(KIND=16) argument, the domain shall be *X* < 11356.5Q0 .

For a COMPLEX(KIND=4) argument, the domain shall be REAL(*X*) < 88.722E0 and ABS(AIMAG(*X*)) < 8.23E+05 .

For a COMPLEX(KIND=8) argument, the domain shall be DREAL(*X*) < 709.782D0 and DABS(DIMAG(*X*)) < 3.53D+15 .

For a COMPLEX(KIND=16) argument, the domain shall be QREAL(*X*) < 11356.5Q0 and QABS(QIMAG(*X*)) < $2.0Q0^{62} * \pi$.

The generic name, EXP, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
a = exp(2.0)
```

2.165 EXP10 Intrinsic Function

Description

Exponential, base 10.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
EXP10	---	1	REAL	REAL
	EXP10		Single-prec REAL	Single-prec REAL
	DEXP10		Double-prec REAL	Double-prec REAL
	QEXP10		Quad-prec REAL	Quad-prec REAL

```
result = EXP10 ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type as *X*. Its value is a REAL representation of 10.0^{x}.

Remarks

EXP10, DEXP10, and QEXP10 evaluate the base 10 exponential of real data, 10.0^{x}.

For a REAL(KIND=4) argument, the domain shall be $X < 38.531E0$.

For a REAL(KIND=8) argument, the domain shall be $X < 308.254D0$.

For a REAL(KIND=16) argument, the domain shall be $X < 4932.0625Q0$.

The generic name, EXP10, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
a = exp10(2.0)  ! a is assigned the value 100.0
```

2.166 EXP2 Intrinsic Function

Description

Exponential, base 2.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
EXP2	---	1	REAL	REAL
	EXP2		Single-prec REAL	Single-prec REAL
	DEXP2		Double-prec REAL	Double-prec REAL
	QEXP2		Quad-prec REAL	Quad-prec REAL

```
result = EXP2 ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type as *X*. Its value is a REAL representation of 2.0^{x}.

Remarks

EXP2, DEXP2, and QEXP2 evaluate the base 2 exponential of real data, 2.0^{x}.

For a REAL(KIND=4) argument, the domain shall be $X < 128.0E0$.

For a REAL(KIND=8) argument, the domain shall be $X < 1024.0D0$.

For a REAL(KIND=16) argument, the domain shall be $X < 16384.0Q0$.

The generic name, EXP2, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
a = exp2(2.0)  ! a is assigned the value 4.0
```

2.167 EXPONENT Intrinsic Function

Description

Exponent part of the model representation of a number.

Class

Elemental function.

Syntax

result = EXPONENT (*X*)

Argument(s)

X

X shall be of type REAL.

Result

The result is of type default INTEGER. Its value is the value of the exponent part of the model representation of *X*.

If *X* is IEEE_INFINITE or IEEE_NAN, the result value is the same as HUGE(0).

Example

```
i = exponent(3.8)   ! i is assigned 2
i = exponent(-4.3)  ! i is assigned 3
```

2.168 EXTENDS_TYPE_OF Intrinsic Function

Description

The EXTENDS_TYPE_OF intrinsic function enquires whether or not the dynamic type is that extended type.

Class

Inquiry function.

Syntax

result = EXTENDS_TYPE_OF (*A* , *MOLD*)

Argument(s)

A

Must be an object of a type that can be extended. If it is a pointer, the allocation status cannot be undefined.

MOLD

Must be an object of a type that can be extended. If it is a pointer, the allocation status cannot be undefined.

Result

The result is of a default LOGICAL scalar.

If *MOLD* is unlimited polymorphic, or if it is either a disassociated pointer or an unallocated allocatable, the result is true.

If *MOLD* is not unlimited polymorphic, and if *A* is either unlimited polymorphic, a disassociated pointer, or has an allocation status of allocatable is unallocated, the result is false.

If the *A* dynamic type is an extension type of the *MOLD* dynamic type, the result is true.

The dynamic type for a disassociated pointer or an allocation status of variable is unallocated, that is the declared type.

Example

```
type ty1
  integer :: i
end type
type, extends(ty1) :: ty2
end type
class (ty1),pointer :: p1,p2
  ...
print *,extends_type_of(p1, p2)
end
```

2.169 EXTERNAL Statement

Description

The EXTERNAL statement specifies external procedures, dummy procedures, or block data program units. Specifying a procedure name as EXTERNAL permits the name to be used as an actual argument.

Syntax

```
EXTERNAL [ :: ] external-name-list
```

Where:

external-name-list is a comma-separated list of external procedures, dummy procedures, or block data program units.

Remarks

If an intrinsic procedure name appears in an EXTERNAL statement, the intrinsic procedure is not available in the scoping unit and the name is that of an external procedure.

Example

```
subroutine fred (a, b, sin)
external sin   ! sin is the name of an external
               ! procedure, not the intrinsic sin()
call bill (a, sin)
               ! sin can be passed as an actual arg
```

2.170 FDATE Service Subroutine

Description

Return the time and date in an ASCII string.

Syntax

```
CALL  FDATE ( string )
```

Argument(s)

string

Default CHARACTER scalar with length greater than or equal to 24.

If the length of argument *string* is greater than 24, blank characters are supplied.

Example

```
use service_routines,only:fdate
character(len=24) :: ch
call fdate(ch)
print *,ch
end
```

2.171 FGETC Service Function

Description

Reads the next available character from a formatted sequential access file connected to the *unit*.

Syntax

iy = FGETC (*unit* , *ch*)

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a formatted sequential access file.

ch

Default CHARACTER(1). Variable whose next available character is to be read from the file.

Result

Default INTEGER scalar. Zero if the read is successful, or -1 if an end-of-file is detected. A positive value is either a Fortran runtime message number.

Example

```
 use service_routines,only:fgetc
 character(len=100) :: ch1
 character(len=10),dimension(10) :: ch
 data ch/"a123456789","b123456789","c123456789","d123456789", &
&        "e123456789","f123456789","g123456789","h123456789", &
&        "i123456789","j123456789"/
 open(10,file='x.dat',form='formatted')
 do i=1,10
   write(10,fmt='(a10)') ch(i)
 end do
 rewind(10)
 do
   if (fgetc(10,ch1(i:i)) .eq. -1) exit
   if (ch1(i:i) .eq. '\n') i=i-1
 end do
 end
```

2.172 FINAL Statement

Description

The FINAL statement defines the final binding for the derived type.

Syntax

`FINAL [::] final-subroutine-name-list`

Where:

The final binding subroutine name must be the name of a module procedure with one dummy argument.

The argument cannot be omitted, and must be a variable that is not a pointer of the derived type being defined, an allocate variable, or a polymorphic variable. The dummy argument cannot be INTENT(OUT).

The same final binding subroutine name cannot be specified more than once for the same type.

A final binding subroutine cannot specify the same kind type parameter and dummy argument with the same rank more than once for one type.

2.173 FLOOR Intrinsic Function

Description

Greatest INTEGER less than or equal to a number.

Class

Elemental function.

Syntax

```
result = FLOOR ( A [ , KIND ] )
```

Required Argument(s)

A

A shall be of type REAL.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is of type default INTEGER. Its value is equal to the greatest INTEGER less than or equal to *A*.

If its value is over capabilities for the specified kind type parameter *KIND*, the result value is undefined.

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

Example

```
i = floor(-2.1)  ! i is assigned the value -3
j = floor(2.1)   ! j is assigned the value 2
```

2.174 FLUSH Service Subroutine

Description

Writes the contents of an external *unit* buffer into its associated file.

Syntax

```
CALL  FLUSH ( unit )
```

Argument(s)

unit

Default INTEGER scalar. Unit number.

Example

```
use service_routines,only:flush,system
open(10,file='x.dat')
do i=1,15
  write(10,*)i
  call flush(10)
  ix = system('cat x.dat')
end do
end
```

2.175 FLUSH Statement

Description

The following items are effective with execution of a FLUSH statement.

- A data written in an external file can be used by other processing.
- A data in an external file excluding Fortran can be used by a READ statement.

A FLUSH statement is not effective for the file position.

You may use a FLUSH statement to a file that is non-being and connected, and it has no effect to any files.

Syntax

```
FLUSH external-file-unit      or
FLUSH ( flush-spec-list )
```

Where:

external-file-unit must be scalar integer expression.

flush-spec-list is a comma-separated list of

```
[ UNIT = ] external-file-unit      or
IOSTAT = io-stator                 or
IOMSG = iomsg                      or
ERR = err-label
```

flush-spec-list shall contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

external-file-unit is an external file device and must be a scalar integer expression.

If the optional characters UNIT= are omitted from the *external-file-unit* specifier, the *external-file-unit specifier* shall be the first item in the *flush-spec-list*.

io-stat must be a variable of type INTEGER that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, and zero otherwise.

iomsg must be a scalar default CHARACTER variable. If the error condition occurs during input/output statement execution, a message for explanation purposes is assigned to *iomsg*.

The *iomsg* value does not change if no error conditions occur.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the FLUSH statement. If an error condition occurs during execution of the FLUSH statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

Example

```
flush 10            ! Data on the buffer of the file connected
                    ! in unit number 10 is output.
flush (10, err=100) ! Data on the buffer of the file connected
                    ! in unit number 10 is output.
                    ! If it is detected in error condition, the statement
                    ! that has a statement number 100 will be executed
                    ! in next.
```

2.176 FORALL Construct

Description

The FORALL construct controls multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements.

Syntax

```
[ forall-construct-name : ] FORALL forall-header
    [ forall-body-construct ] ...
END FORALL [ forall-construct-name ]
```

Where:

forall-construct-name is an optional name for the FORALL construct name.

If the FORALL construct statement has a *forall-construct-name*, the END FORALL statement shall have the same *forall-construct-name*. If the END FORALL statement has a *forall-construct-name*, the FORALL construct statement shall have the same *forall-construct-name*.

See "1.16 FORALL Header" for *forall-header*.

forall-body-construct is

```
assignment statement                or
pointer assignment statement        or
WHERE statement                     or
WHERE construct                     or
FORALL construct                    or
FORALL statement
```

A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.

Any procedure referenced in a *forall-body-construct* shall be a pure procedure, it is including one referenced by a defined operation, assignment, or finalization.

Remarks

When the FORALL construct is executed, the *subscript* and *stride* expressions in the *forall-triplet-spec-list* are evaluated. The set of values that a particular *index-name* variable assumes is determined the expression $m_1 + (k\text{-}1) * m_3$, where k is k=1, 2, ..., *max* , and *max* is $(m_2 - m_1 + m_3)/m_3$. The m_1, m_2, and m_3 are established by evaluating the first *subscript*, the second *subscript*, and the *stride* expressions. If a *stride* does not appear, m_3 hat the value 1. If *max* <= 0 for some *index-name*, the FORALL construct is not executed.

The *scalar-mask-expr*, if any, is evaluated for each combination of *index-name* values. If the *scalar-mask-expr* is not present, it is as if it were present with the value .TRUE.. The active combination of *index-name* values is defined to be the subset of all possible combinations for which the *scalar-mask-expr* has the value .TRUE..

The *forall-body-construct*s are executed in the order in which they appear.

Execution of an assignment statement in a FORALL construct causes the evaluation of expression on the right side of the equal sign and all expressions within the variable on the left side of the equal sign. After all these evaluations have been performed, each expression value on the right side of the equal sign is assigned to the corresponding variable on the left side of the equal sign.

Execution of a pointer assignment statement in a FORALL construct causes the evaluation of all expressions within target and pointer object. After all these evaluations have been performed, each pointer object is associated with the corresponding target.

Each statement in a WHERE construct within a FORALL construct is executed in sequence. When a WHERE statement, WHERE construct statement, or masked ELSEWHERE statement is executed, the statement's *mask-expr* is evaluated for all active combinations of *index-name* values as determined by the outer FORALL constructs, masked by any control mask corresponding to outer WHERE constructs. Any assignment statement in a WHERE construct is executed for all active combinations of *index-name* values, masked by the control mask in effect for the assignment statement. See "2.486 WHERE Construct" for WHERE construct, and "2.488 WHERE Statement" for WHERE statement.

Execution of a FORALL statement or FORALL construct in other FORALL construct causes the evaluation of the *subscript* and *stride* expressions in the *forall-triplet-spec-list* for all active combinations of the *index-name* values of the outer FORALL construct. The set of combinations of *index-name* values for the inner FORALL is the union of the sets defined by these bounds and strides for each active combination of the outer *index-name* values; it also includes the outer *index-name* values. The *scalar-mask-expr* is then evaluated for all combinations of the *index-name* values of the inner construct to produce a set of active combinations for the inner FORALL construct. If there is no *scalar-mask-expr*, it is as if it were present with the value .TRUE.. Each statement in the inner FORALL is then executed for each active combination of the *index-name* values.

The *forall-body-construct*s are executed in the order in which they appear for all active combination of *index-name* values, that is not same as the DO construct. If the *forall-body-construct*s has more than one statements, please note the variable which is defined in the previous statements is evaluated and defined for all active combination of *index-name* values.

Example

```
integer :: a(3,3)
integer :: n=3
a = reshape((/0,1,2,3,4,5,6,7,8/),(/3,3/))
                !  represents  |0 3 6|
                !              |1 4 7|
                !              |2 5 8|
forall (i=1:n-1)
  forall (j=i+1:n)
    a(i,j) = a(j,i)
  end forall
end forall
                !  represents  |0 1 2|
                !              |1 4 5|
                !              |2 5 8|
```

2.177 FORALL Construct Statement

Description

The FORALL construct statement begins a FORALL construct. See "2.176 FORALL Construct" for FORALL construct.

Syntax

`[ forall-construct-name : ] FORALL forall-header`

Where:

forall-construct-name is an optional name for the FORALL construct name.

See "1.16 FORALL Header" for *forall-header*.

2.178 FORALL Statement

Description

The FORALL statement controls the execution an assignment or pointer assignment statement with selection by sets of index values and an optional mask expression.

Syntax

`FORALL forall-header  forall-assignment-stmt`

Where:

See "1.16 FORALL Header" for *forall-header*.

forall-assignment-stmt is

```
assignment-stmt             or
pointer-assignment-stmt
```

Remarks

A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct* that is a *forall-assignment-stmt*. See "2.176 FORALL Construct" for FORALL construct.

2.179 FORK Service Function

Description

Creates a copy of the calling process.

Syntax

```
iy = FORK ( )
```

Result

Default INTEGER scalar. A child process id if the creation is successful; otherwise, a negation of a system error code.

Example

```
      use service_routines,only:fork
      integer :: pid
      pid = fork()
      if (pid) 10,20,30
10    write(6,fmt="(5x,a)") "FORK FAILED"
      stop 240
20    write(6,fmt="(3x,a)") "CHILD PROCESS RUNNING  // FORK SUCCEEDED //"
30    stop
      end
```

2.180 FORMAT Statement

Description

The FORMAT statement provides explicit information that directs the editing between the internal representation of data and the characters that are input or output.

Syntax

```
FORMAT format-specification
```

Where:

format-specification is a format specification (see "1.8.1 Format Specification").

Remarks

The FORMAT statement shall be labeled.

Example

```
10 format (e11.5)
20 format (3i8, e12.5)
```

2.181 FPUTC Service Function

Description

Writes a character to a formatted sequential access file that is connected to the *unit*.

Syntax

```
iy = FPUTC ( unit , ch )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a formatted sequential access file.

ch

Default CHARACTER(1). A character to be written to the file.

Result

Default INTEGER scalar. The value of each function is zero if successful, and -1 or a Fortran runtime message number otherwise.

Example

```
use service_routines,only:fputc
character(len=6) :: pr='INPUT>'
do i=1,6
  if (fputc(6,pr(i:i)) .ne. 0) stop 10
end do
read (*,*)x
end
```

2.182 FRACTION Intrinsic Function

Description

Fraction part of the physical representation of a number.

Class

Elemental function.

Syntax

```
result = FRACTION ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same kind as *X*.
Its value is the value of the fraction part of the physical representation of *X*; *X* * RADIX(*X*)$^{(-EXPONENT(X))}$.

When *X* is 0.0, the result value is 0.0. If *X* is IEEE infinity, the result is infinity. If *X* is IEEE NaN, the result is NaN.

Example

```
a = fraction(3.8) ! a is assigned the value 0.95
```

2.183 FREE Service Subroutine

Description

Frees a block of memory that is currently allocated.

Syntax

```
CALL  FREE ( addr )
```

Argument(s)

addr

Eight-byte INTEGER scalar. Starting address of the memory block to be freed, previously allocated by MALLOC Service Function.

If the freed address was not previously allocated with MALLOC, or if an address is freed more than once, the results are undefined.

Example

```
use service_routines,only:free,malloc
integer(8) :: i
i = malloc(20_8)
  ...
```

```
call free(i)
end
```

2.184 FSEEK Service Function

Description

Repositions a sequential access file-position indicator in the *unit*.

Syntax

```
iy = FSEEK ( unit , offset , from )
```

Argument(s)

unit

Default INTEGER scalar. *unit* is unit number . If the unit number is not connected to a file, it will be connected to the formatted sequential access.

offset

Default INTEGER scalar. Offset in bytes, relative to from, that is to be the new position of the file.

from

Default INTEGER scalar. A position in the file. The values specified in FSEEK as follows:

```
=0 : Beginning of the file
=1 : Current position
=2 : End of the file
```

Result

Default INTEGER scalar. This function returns 0 on success; otherwise, it returns the value which indicates Fortran runtime message number.

Example

```
    use service_routines,only:fseek
    open(10,file='x.dat',status='replace')
    do i=1,50
      write(10,*) i
    end do
    close(10)
    open(10,file='x.dat')
    i = fseek(10,0,2)
    write(10,*) 51
    i = fseek(10,0,0)
    do
      read(10,*,end=30) i
    end do
    go to 40
30  print *,'EOF',i
40  close(10)
    end
```

2.185 FSEEKO64 Service Function

Description

Repositions a sequential access file-position indicator in the *unit*.

Syntax

```
iy = FSEEKO64 ( unit , offset , from )
```

Argument(s)

unit

Default INTEGER scalar. *unit* is unit number . If the unit number is not connected to a file, it will be connected to the formatted sequential access.

offset

Eight byte INTEGER scalar. Offset in bytes, relative to from, that is to be the new position of the file.

from

Default INTEGER scalar. A position in the file. The values specified in FSEEKO64 as follows:

```
=0 : Beginning of the file
=1 : Current position
=2 : End of the file
```

Result

Default INTEGER scalar. This function returns 0 on success; otherwise, it returns the value which indicates Fortran runtime message number.

Example

```
   use service_routines,only:fseeko64
   open(10,file='x.dat',status='replace')
   do i=1,50
     write(10,*) i
   end do
   close(10)
   open(10,file='x.dat')
   i = fseeko64(10,0_8,2)
   write(10,*) 51
   i = fseeko64(10,0_8,0)
   do
     read(10,*,end=30) i
   end do
   go to 40
30 print *,'EOF',i
40 close(10)
   end
```

2.186 FSTAT Service Function

Description

Returns information about the file connected to the *unit*.

Syntax

```
iy = FSTAT ( unit , status )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a file.

status

Default INTEGER and rank one. Its size shall be at least 13. The values returned in *status* are as follows:

```
status(1)  : File mode.
status(2)  : Inode number.
status(3)  : ID of device containing a directory entry for this file.
status(4)  : ID of device. This entry is defined only for char special or block special files.
status(5)  : Number of links.
```

```
status(6)  : User ID of the file's owner.
status(7)  : Group ID of the file's group.
status(8)  : File size in bytes.
status(9)  : Time of last access.
status(10) : Time of last data modification.
status(11) : Time of last file status change.
status(12) : Preferred I/O block size.
status(13) : Number of 512 byte blocks allocated.
```

Result

Default INTEGER scalar. Zero if successful, or 114 if the file is closed. Otherwise, returns a system error code.

Example

```
use service_routines,only:fstat
integer :: st(13)
print *,fstat(10,st)
end
```

2.187 FSTAT64 Service Function

Description

Returns information about the file connected to the *unit*.

Syntax

```
iy = FSTAT64 ( unit , status )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a file.

status

Eight byte INTEGER and rank one. Its size shall be at least 13. The values returned in *status* are as follows:

```
status(1)  : File mode.
status(2)  : Inode number.
status(3)  : ID of device containing a directory entry for this file.
status(4)  : ID of device. This entry is defined only for char special or block special files.
status(5)  : Number of links.
status(6)  : User ID of the file's owner.
status(7)  : Group ID of the file's group.
status(8)  : File size in bytes.
status(9)  : Time of last access.
status(10) : Time of last data modification.
status(11) : Time of last file status change.
status(12) : Preferred I/O block size.
status(13) : Number of 512 byte blocks allocated.
```

Result

Default INTEGER scalar. Zero if successful, or 114 if the file is closed. Otherwise, returns a system error code.

Example

```
use service_routines,only:fstat64
integer(kind=8) :: st(13)
print *,fstat64(10,st)
end
```

2.188 FTELL Service Function

Description

Returns the current position of a formatted sequential access file connected to the *unit*.

Syntax

```
iy = FTELL ( unit )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a formatted sequential access file.

Result

Default INTEGER scalar. Offset, in bytes, from the beginning of a formatted sequential access file.

A negative value indicates an error.

Example

```
use service_routines,only:ftell,fseek
integer :: ix(10)
open(10,file='x.dat',status='replace')
do i=1,10
  write(10,*) i
  ix(i) = ftell(10)
  if(ix(i) < 0) exit
end do
close(10)
open(10,file='x.dat')
i = fseek(10,ix(3),0)
if (i < 0) stop
read(10,*) i
if (i /= 4) stop 'error'
close(10)
end
```

2.189 FTELLO64 Service Function

Description

Returns the current position of a formatted sequential access file connected to the *unit*.

Syntax

```
iy = FTELLO64 ( unit )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a formatted sequential access file.

Result

Eight byte INTEGER scalar. Offset, in bytes, from the beginning of a formatted sequential access file.

A negative value indicates an error.

Example

```
use service_routines,only:ftello64,fseeko64
integer(kind=8) :: ix(10),i
open(10,file='x.dat',status='replace')
do i=1,10
  write(10,*) i
```

```
  ix(i) = ftello64(10)
  if(ix(i) < 0) exit
end do
close(10)
open(10,file='x.dat')
i = fseeko64(10,ix(3),0)
if (i < 0) stop
read(10,*) i
if (i /= 4) stop 'error'
close(10)
end
```

2.190 FUNCTION Statement

Description

The FUNCTION statement begins a function subprogram, and specifies characteristics of its function result. See "1.12.1 Function Subprogram" for function subprogram.

Syntax

```
[ prefix-spec ] ... FUNCTION function-name ( [ dummy-arg-list ] ) [ suffix ]
```

Where:

prefix-spec is

```
type-spec        or
RECURSIVE        or
PURE             or
ELEMENTAL
```

type-spec is a declaration type specifier that is the following syntax:

```
intrinsic-type-spec          or
TYPE ( type-name )           or
CLASS( type-name )           or
CLASS( * )
```

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
BYTE
```

kind-selector is

```
( [ KIND = ] kind )        or
* mem-length
```

char-selector is

```
( LEN = char-length-parm , KIND = kind )         or
( char-length-parm , [ KIND = ] kind )           or
( KIND = kind , LEN = char-length-parm )         or
( [ LEN = ] char-length-parm )                   or
* char-length  (obsolescent feature )
```

char-length is

```
( char-length-parm )             or
scalar-int-literal-constant
```

char-length-parm is

```
specification-expr          or
*                           or
:
```

kind is the value of kind type parameter. *kind* is a scalar INTEGER initialization expression.

mem-length is a number of bytes used to represent each respective type. It shall be a scalar INTEGER initialization expression. If the *type-spec* is INTEGER, REAL, or LOGICAL, the value of *mem-length* has the same meaning as the value of *kind*. If the *type-spec* is COMPLEX, the value of *mem-length* has the same meaning as the value of *kind**2.

See "2.469 Type Declaration Statement" for the value of *kind* and *mem-length*.

A character length parameter value of ':' indicates a deferred type parameter.

A function name that is the name of internal or module function, or specified in interface body shall not be declared with an asterisk *char-length-parm*.

type-name is the name of the derived type. *type-name* is defined within the body of the function or is accessible there by use or host association.

The type and type parameters of the result of the function defined by a function subprogram may be specified by a type specification in the FUNCTION statement or by the name of the result variable appearing in a type declaration statement in the declaration part of the function subprogram. It shall not be specified both ways.

function-name is the name of the function. If RESULT is not specified, the result variable is *function-name*.

dummy-arg-list is a comma-separated list of

```
dummy-arg-name
```

dummy-arg-name is the name of the dummy argument.

suffix is

```
proc-language-binding-spec [ RESULT ( result-name ) ]         or
RESULT ( result-name ) proc-language-binding-spec
```

result-name is the name of the result variable.

proc-language-binding-spec is the following procedure language binding specifier.

```
BIND ( C [, NAME = scalar-char-initialization-expr ]
```

Remarks

If the RECURSIVE is present, the function subprogram is a recursive procedure. If the PURE is present, the function subprogram is a pure subprogram. If the ELEMENTAL is present, the function subprogram is an elemental subprogram. See "1.12.3 Recursive Procedures" for recursive procedure, "1.12.4 Pure Procedures" for pure subprogram, "1.12.5 Elemental Procedures" for elemental subprogram.

If RESULT is specified, *result-name shall* not be the same as *function-name*. If RESULT is specified, the name of the result variable of the function is *result-name*, and all occurrences of the function name in execution part statements in the scoping unit are recursive function references. If RESULT is not specified, the result variable is *function-name* and all occurrences of the function name in execution part statements in the scoping unit are references to the result variable.

The CLASS specifier can be used to declare a polymorphic object. If the CLASS specifier includes a type name, that type becomes the declared type of the polymorphic object.

An object declared with the CLASS(*) specifier is unlimited polymorphic. An unlimited polymorphic data entity does not have a declared type. It is treated as having a declared type different to that of all other entities.

Example

```
function sum(i,j) result(k)
  integer :: i,j,k
  ...
end function
```

2.191 GAMMA Intrinsic Function

Description

Gamma.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
GAMMA	---	1	REAL	REAL
	GAMMA		Single-prec REAL	Single-prec REAL
	DGAMMA		Double-prec REAL	Double-prec REAL
	QGAMMA		Quad-prec REAL	Quad-prec REAL
LGAMMA	---	1	REAL	REAL
	LGAMMA		Single-prec REAL	Single-prec REAL
	ALGAMA		Single-prec REAL	Single-prec REAL
	DLGAMA		Double-prec REAL	Double-prec REAL
	QLGAMA		Quad-prec REAL	Quad-prec REAL

```
result = GAMMA ( X )
result = LGAMMA ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type as *X*. Its value is a REAL representation of the gamma function of data of type REAL.

Remarks

GAMMA, DGAMMA, and QGAMMA evaluate the gamma function of data of type REAL.

For a REAL(KIND=4) argument, the domain shall be 0.0E0 < *X* < 35.039890E0.

For a REAL(KIND=8) argument, the domain shall be 0.0D0 < *X* < 171.6243D0.

For a REAL(KIND=16) argument, the domain shall be 0.0Q0 < *X* < 1.755Q+03.

LGAMMA, ALGAMA, DLGAMA, and QLGAMA evaluate the logarithmic gamma function of data of type REAL.

For a REAL(KIND=4) argument, the domain shall be 0 < *X* < 4.03711E+36.

For a REAL(KIND=8) argument, the domain shall be 0 < *X* < 2.55634D+305.

For a REAL(KIND=16) argument, the domain shall be 0 < *X* < 1.048Q+4928.

The generic names, GAMMA and LGAMMA, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
a = gamma(.5)
a = lgamma(.5)
```

2.192 GETARG Service Subroutine

Description

Returns the specified command-line argument (where the command itself is argument number 0).

Syntax

```
CALL  GETARG ( argno , argst )
```

Argument(s)

argno

Default INTEGER scalar.

argst

Default CHARACTER scalar. Variable receives command-line argument.

Remarks

If the command-line argument is shorter than *argst*, GETARG pads *argst* on the right with blanks. If the argument is longer than *argst*, GETARG truncates the argument.

Example

```
use service_routines,only:getarg
integer :: i=1
character(len=10) :: ch
call getarg(i,ch)
print *,'ch=',ch
end
```

2.193 GETC Service Function

Description

Reads the next available character from the standard input file.

Syntax

```
iy = GETC ( ch )
```

Argument(s)

ch

Default CHARACTER(1). Variable whose value is set to the next available character read from the standard input file.

Result

Default INTEGER scalar. Zero if the read is successful, or -1 if an end-of-file is detected. A positive value is either a Fortran runtime message number.

Example

```
use service_routines,only:getc
character(len=10) :: ch1
ch1 = ' '
do i=1,10
  if (getc(ch1(i:i)) == -1 ) exit
end do
call prompt(ch1)
read (*,*) i
end
```

2.194 GETCL Service Subroutine

Description

Returns the specified command-line argument.

Syntax

```
CALL  GETCL ( string )
```

Argument(s)

string

Default CHARACTER scalar. Variable receives the command-line argument.

Remarks

If the command-line argument is shorter than string, GETCL pads string on the right with blanks. If the argument is longer than string, GETCL truncates the argument.

Example

```
use service_routines,only:getcl
character(len=15) :: ch
call getcl(ch)
print *,'argument line=',ch
end
```

2.195 GETCWD Service Function

Description

Returns the current directory pathname.

Syntax

```
iy = GETCWD ( ch )
```

Argument(s)

ch

Default CHARACTER scalar. The current directory name. If the pathname is shorter than *ch*, GETCWD pads *ch* on the right with blanks. If the pathname is longer than *ch*, GETCWD truncates the pathname.

Result

Default INTEGER scalar. Zero if successful, or -1 if the pathname is longer than *ch*. Otherwise, a system error code.

Example

```
use service_routines,only:getcwd
integer :: iy
character(len=40) ch
iy = getcwd(ch)
end
```

2.196 GETDAT Service Subroutine

Description

Returns the date.

Syntax

```
CALL  GETDAT ( year , month , day )
```

Argument(s)

year

Two-byte INTEGER, scalar. Year (xxxx AD).

month

Two-byte INTEGER, scalar. Month (1-12).

day

Two-byte INTEGER, scalar. Day of the month (1-31).

Example

```
use service_routines,only:getdat
integer(2) :: year,month,date
call getdat(year,month,date)
write (6,fmt="(1x,i4,1x,i2,1x,i2)") year,month,date
end
```

2.197 GETENV Service Subroutine

Description

Returns the value of an environment variable.

Syntax

```
CALL  GETENV ( env , string )
```

Argument(s)

env

Default CHARACTER scalar. Environment variable.

string

Default CHARACTER scalar. Value of *env*.

Remarks

If the value of an environment variable is shorter than *string*, GETENV pads *string* on the right with blanks. If the value is longer than *string*, GETENV truncates the value. String is set to blanks if the environment variable *env* is not set.

Example

```
use service_routines,only:getenv
character(len=10) :: ch
call getenv('FORT90L',ch)
print *,'FORT90L=',ch
end
```

2.198 GETFD Service Function

Description

Returns the file descriptor of a *unit* connected to a file.

Syntax

```
iy = GETFD ( unit )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a file.

Result

Default INTEGER scalar. GETFD returns the file descriptor of an external unit number if the file that connected with an external unit number is opened, and -1 otherwise.

Example

```
use service_routines,only:getfd
if (getfd(10) == getfd(11)) stop 'error'
end
```

2.199 GETGID Service Function

Description

Retrieves the group ID of the user of a process.

Syntax

```
iy = GETGID ( )
```

Result

Default INTEGER scalar. The group ID.

Example

```
use service_routines,only:getgid
print *,getgid()
end
```

2.200 GETLOG Service Subroutine

Description

Returns the user's login name.

Syntax

```
CALL  GETLOG ( name )
```

Argument(s)

name

Default CHARACTER scalar. User's login name.

If the login name is shorter than name, GETLOG pads the name on the right with blanks. If the login name is longer than name, GETLOG truncates the name.

Example

```
use service_routines,only:getlog
character(len=10) :: ch
call getlog(ch)
print *,'login name=',ch
end
```

2.201 GETPARM Service Subroutine

Description

Returns the specified command-line argument.

Syntax

```
CALL  GETPARM ( len , parm )
```

Argument(s)

len

Default INTEGER scalar. The length of command-line argument.

parm

Default CHARACTER scalar. It receives the command-line argument.

Remarks

If the command-line argument is shorter than *parm*, GETPARM pads *parm* on the right with blanks. If the argument is longer than *parm*, GETPARM truncates the argument.

Example

```
use service_routines,only:getparm
integer :: leng
character(len=10) :: parm
call getparm(leng,parm)
write(6,*) 'length=',leng,',argument=',parm
end
```

2.202 GETPID Service Function

Description

Returns the process ID of the current process.

Syntax

```
iy = GETPID ( )
```

Result

Default INTEGER scalar. The process ID of the current process.

Example

```
use service_routines,only:getpid
print *,getpid()
end
```

2.203 GETTIM Service Subroutine

Description

Return the time.

Syntax

```
CALL  GETTIM ( hour , minutes , second , second1_100 )
```

Argument(s)

hour

Two-byte INTEGER, scalar. Hour (0-23).

minutes

Two-byte INTEGER, scalar. Minute (0-59).

second

Two-byte INTEGER, scalar. Second (0-59).

second1_100

Two-byte INTEGER, scalar. Hundredths of a second (0-99).

Example

```
use service_routines,only:gettim
integer(kind=2) :: h,m,s,s1_100
call gettim(h,m,s,s1_100)
write (6,fmt="(1x,i2,':',i2,':',i2,'.',i2)") h,m,s,s1_100
end
```

2.204 GETTOD Service Subroutine

Description

The GETTOD service subroutine returns the current high-resolution real time, expressed as microseconds since the some arbitrary time in the past. Typically, the time is expressed since the most recent system booting.

Syntax

```
CALL GETTOD ( g )
```

Argument(s)

g

Double precision REAL scalar. Elapsed time expressed as microseconds is set to *g*.

Example

```
use service_routines,only:gettod
real(kind = 8) :: g0, g1
call gettod(g0)
call sub
call gettod(g1)
write(6, *) g1 - g0, '(microsec)'
end
```

2.205 GETUID Service Function

Description

Retrieves the user ID of the calling process.

Syntax

```
iy = GETUID ( )
```

Result

Default INTEGER scalar. The user ID.

Example

```
use service_routines,only:getuid
print *,getuid()
end
```

2.206 GET_COMMAND Intrinsic Subroutine

Description

The intrinsic subroutine GET_COMMAND returns entire command when the program is called.

Class

Subroutine.

Syntax

```
CALL GET_COMMAND ( [ COMMAND, LENGTH, STATUS ] )
```

Optional Argument(s)

COMMAND

COMMAND must be scalar of default CHARACTER type and INTENT(OUT) argument.

COMMAND is set entire command when the program is called.

If command does not exist, *COMMAND* is set by blanks at all.

LENGTH

LENGTH must be scalar of default INTEGER type and INTENT(OUT) argument.

LENGTH is assigned the significant length of the command when the program is called. The significant length of the command does not include following blanks. The command length is set a valid command length regardless of a length in the argument *COMMAND*.

Also, the command length is set a valid command length when the argument *COMMAND* is omitted. If the command does not exist, *LENGTH* is zero.

STATUS

STATUS must be scalar of default INTEGER type and INTENT(OUT) argument. If you specify an argument *COMMAND*, and length is smaller than length of effective command, it is set to -1. If command does not exist, *STATUS* is set to 1. Otherwise *STATUS* is set to zero.

Example

```
character(len=30) :: string
integer :: len
call get_command(string, len)  ! % a.out arg1 arg2 arg3
  ! The variable string is set to command "a.out arg1 arg2 arg3",
  ! the variable len is set to length of command twenty when the program
  ! is called.
```

2.207 GET_COMMAND_ARGUMENT Intrinsic Subroutine

Description

The intrinsic subroutine GET_COMMAND_ARGUMENT returns command argument when the program is called.

Class

Subroutine.

Syntax

```
CALL GET_COMMAND_ARGUMENT ( NUMBER [ , VALUE, LENGTH, STATUS ] )
```

Required Argument(s)

NUMBER

NUMBER shall be scalar of default INTEGER type and an INTENT(IN) argument. It specifies the number of the command argument that the other arguments give information about. Useful values of *NUMBER* are those between zero and the argument count returned by the COMMAND_ARGUMENT_COUNT intrinsic function. Other values are allowed, but will result in error status return.

Command argument zero is defined to be the command name by which the program was invoked. The remaining command arguments are numbered consecutively from one to the argument count in an order determined by the processor.

Optional Argument(s)

VALUE

VALUE shall be scalar and of default CHARACTER type and an INTENT(OUT) argument. It is assigned to the value of the command argument specified by *NUMBER*. If the command argument value cannot be determined, *VALUE* is assigned all blanks.

LENGTH

LENGTH shall be scalar and of default INTEGER type and an INTENT(OUT) argument. It is assigned the significant length of the command argument specified by *NUMBER*. The significant length may not include trailing blanks if the processor allows command arguments with significant trailing blanks. This length is set a valid command length regardless of a length in the argument *VALUE*. Also, the length is set a valid command length when the argument *VALUE* is omitted. If the command argument length cannot be determined, a length of zero is assigned.

STATUS

STATUS shall be scalar and of default INTEGER type and 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 the value that is number of command arguments and command name. Otherwise it is assigned the value zero.

Example

```
character(len=30) :: string
integer :: pos, len
pos = 2
call get_command_argument(pos, string, len)  ! % a.out arg1 arg2 arg3
  ! The variable string is set to the second command argument 'arg2'.
  ! The variable len is set to length of the second command argument 4.
```

2.208 GET_ENVIRONMENT_VARIABLE Intrinsic Subroutine

Description

The intrinsic subroutine GET_ENVIRONMENT_VARIABLE gets the value of an environment variable.

Class

Subroutine.

Syntax

```
CALL GET_ENVIRONMENT_VARIABLE ( NAME [ , VALUE, LENGTH, STATUS, TRIM_NAME ] )
```

Required Argument(s)

NAME

NAME shall be a scalar and default CHARACTER type and an INTENT(IN) argument. The name of environment variable shall be specified in *NAME*.

Optional Argument(s)

VALUE

VALUE shall be a scalar of default CHARACTER type and an INTENT(OUT) argument. It is assigned to 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.

LENGTH

LENGTH shall be a scalar of default INTEGER type and 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 zero.

STATUS

STATUS shall be scalar and of default INTEGER type and 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 1 if the specified environment variable does not exist.

TRIM_NAME

TRIM_NAME shall be a scalar of LOGICAL type and an INTENT(IN) argument. If *TRIM_NAME* is specified with the value false then trailing blanks in *NAME* are considered significant. If *TRIM_NAME* is omitted or *TRIM_NAME* is specified with the value true then trailing blanks in *NAME* are ignored.

Example

```
character(len=30) :: env = 'SHELL'
character(len=30) :: string
call get_environment_variable(env, string)
  ! The value of the environment variable SHELL is set in variable string.
```

2.209 GMTIME Service Subroutine

Description

Returns Greenwich mean time in an array of *time* elements.

Syntax

```
CALL  GMTIME ( time , t )
```

Argument(s)

time

Default INTEGER scalar. Numeric time data to be formatted.
Number of seconds since 00:00:00 Greenwich mean time, January 1, 1970.

t

Default INTEGER array of rank one. Its size shall be at least 9.
The values returned in *t* are as follows:

Element	Value
t(1)	Seconds after the minute (0-59)
t(2)	Minutes after the hour (0-59)
t(3)	Hours since midnight (0-23)
t(4)	Day of month (1-31)
t(5)	Month since January (0-11)
t(6)	Year since 1900
t(7)	Days since Sunday (0-6)
t(8)	Days since January (0-365)
t(9)	Daylight saving flag (0 if standard time, 1 if daylight saving time)

Example

```
use service_routines,only:gmtime,time
integer :: t(9)
call gmtime(time(),t)
write(6,fmt="(1x,9i4)") t
end
```

2.210 GO TO Statement

Description

The GO TO statement transfers control to a statement identified by a label.

Syntax

```
GO TO label
```

Where:

label is the label of a branch target statement.

Remarks

label shall be the label of a branch target statement in the same scoping unit as the GO TO statement.

Example

```
      a = b
      go to 10        ! branches to 10
      b = c           ! never executed
10    c = d
```

2.211 HOSTNM Service Function

Description

Retrieves the current host computer name.

Syntax

```
iy = HOSTNM ( name )
```

Argument(s)

name

Default CHARACTER scalar. Name of the current host. If the name is shorter than *name*, HOSTNM pads *name* on the right with blanks. If the name is longer than *name*, HOSTNM truncates the *name*.

Result

Default INTEGER scalar. Zero if successful; otherwise, a system error code.

Example

```
use service_routines,only:hostnm
integer :: iy
character(len=100) :: name
iy = hostnm(name)
print *,'hostname = ',name
end
```

2.212 HUGE Intrinsic Function

Description

Largest representable number of data type.

Class

Inquiry function.

Syntax

```
result = HUGE ( X )
```

Argument(s)

X

X shall be of type REAL or INTEGER. It can be scalar or array-valued.

Result

The result is of the same type and kind as *X*. Its value is the value of the largest number in the data type of *X*.

The result value is as follows:

Type of *X*	The result value
One-byte INTEGER	127_1
Two-byte INTEGER	32767_2
Four-byte INTEGER	2147483647_4
Eight-byte INTEGER	9223372036854775807_8
Single precision REAL	3.40282347E+38_4
Double precision REAL	1.797693134862316E+308_8
Quadruple precision REAL	1.189731495357231765085759326628007OE+4932_16

Example

```
a = huge(4.1)  ! a is assigned the value 3.40282347E+38
```

2.213 HYPOT Intrinsic Function

Description

Euclidean distance.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
HYPOT	---	2	REAL, REAL	REAL

```
result = HYPOT ( X ,Y )
```

Argument(s)

X

X shall be of type REAL.

Y

Y shall be of type REAL of the same kind as *X*.

Result

The result is of the same kind as *X*.

Remarks

HYPOT evaluates the Euclidean distance $\sqrt{(x^2+y^2)}$.

The result is of the same kind as *X*.

Example

```
r = hypot(3.0 , 4.0)
```

2.214 IACHAR Intrinsic Function

Description

Position of a character in the ASCII collating sequence.

Class

Elemental function.

Syntax

```
result = IACHAR ( C [ , KIND ] )
```

Required Argument(s)

C

C shall be of type default CHARACTER and of length one.

Optional Argument(s)

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

The result is of type INTEGER.
Its value is the position of *C* in the ASCII collating sequence and is in the range 0 <= IACHAR(*C*) <= 127 .

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

Example

```
i = iachar('c')  ! i is assigned the value 99
```

2.215 IAND Intrinsic Function

Description

Bit-wise logical AND.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
IAND	---	2	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant	INTEGER
IAND or AND	---	2	One-byte INTEGER, One-byte INTEGER	One-byte INTEGER
	IIAND		Two-byte INTEGER, Two-byte INTEGER	Two-byte INTEGER
	IAND		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	AND		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	JIAND		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER, Eight-byte INTEGER	Eight-byte INTEGER

```
result = IAND ( I ,  J )
result = AND ( I ,  J )
```

Argument(s)

I

I shall be of type INTEGER. IAND may be used a binary,octal or hexadecimal constant.

J

J shall be of type INTEGER and of the same kind as *I*. IAND may be used a binary,octal or hexadecimal constant.

Result

The result is of type INTEGER. Its value is the value obtained by performing a bit-wise logical AND of *I* and *J*.

Remarks

IAND, AND, IIAND, and JIAND evaluate the bit-wise logical AND of type INTEGER.

The generic names, IAND and AND, may be used with any INTEGER argument.

If a binary,octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. The binary, octal or hexadecimal constants for both shall not specify.

The type of the result of each function is the same as the type of the argument *I* or *J*.

Example

```
i=53
j=45
k=iand(i,j)  ! k is assigned the value 37
```

2.216 IARGC Service Function

Description

Returns the number of strings specified by the runtime options and the program parameters.

Syntax

```
iy = IARGC ( )
```

Result

Default INTEGER scalar. The number of the command line arguments.

Example

```
use service_routines,only:iargc
print *,iargc()
end
```

2.217 IBCHNG Intrinsic Function

Description

Reverse one bit.

Class

Elemental function.

Syntax

```
result = IBCHNG ( I , POS )
```

Argument(s)

I

I shall be of type INTEGER.

POS

POS shall be of type INTEGER. It shall be non-negative and less than the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* except that bit *POS* is reversed. Note that the lowest order *POS* is zero.

Example

```
i = ibchng(7,1)  ! i is assigned the value 5
i = ibchng(8,2)  ! i is assigned the value 12
```

2.218 IBCLR Intrinsic Function

Description

Clear one bit to zero.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
IBCLR	---	2	One-byte INTEGER, INTEGER	One-byte INTEGER
	IIBCLR		Two-byte INTEGER, INTEGER	Two-byte INTEGER
	IBCLR		Four-byte INTEGER, INTEGER	Four-byte INTEGER
	JIBCLR		Four-byte INTEGER, INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER, INTEGER	Eight-byte INTEGER

```
result = IBCLR ( I , POS )
```

Argument(s)

I

I shall be of type INTEGER.

POS

POS shall be of type INTEGER. It shall be non-negative and less than the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* except that bit *POS* is set to zero. Note that the lowest order *POS* is zero.

Remarks

IBCLR, IIBCLR, and JIBCLR clear one bit *POS* in *I*.

The generic name, IBCLR, may be used with any INTEGER argument.

The type of the result of each function is the same as the type of the argument.

Example

```
i = ibclr (37,2) ! i is assigned the value 33
```

2.219 IBITS Intrinsic Function

Description

Extract a sequence of bits.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
IBITS	---	3	One-byte INTEGER, INTEGER , INTEGER	One-byte INTEGER
	IIBITS		Two-byte INTEGER, INTEGER , INTEGER	Two-byte INTEGER
	IBITS		Four-byte INTEGER, INTEGER , INTEGER	Four-byte INTEGER
	JIBITS		Four-byte INTEGER, INTEGER , INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER, INTEGER , INTEGER	Eight-byte INTEGER

```
result = IBITS ( I ,  POS , LEN )
```

Argument(s)

I

I shall be of type INTEGER.

POS

POS shall be of type INTEGER. It shall be non-negative and *POS*+*LEN* shall be less than or equal to the number of bits in *I*.

LEN

LEN shall be of type INTEGER and non-negative.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of the sequence of *LEN* bits beginning with *POS*, right adjusted with all other bits set to 0. Note that the lowest order *POS* is zero.

Remarks

IBITS, IIBITS, and JIBITS extract a sequence of *LEN* bits from *POS* in *I*.

The generic name, IBITS, may be used with any INTEGER argument.

The type of the result of each function is the same as the type of the argument.

Example

```
i = ibits (37,2,2) ! i is assigned the value 1
```

2.220 IBSET Intrinsic Function

Description

Set a bit to one.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
IBSET	---	2	One-byte INTEGER, INTEGER	One-byte INTEGER
	IIBSET		Two-byte INTEGER, INTEGER	Two-byte INTEGER
	IBSET		Four-byte INTEGER, INTEGER	Four-byte INTEGER
	JIBSET		Four-byte INTEGER, INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER, INTEGER	Eight-byte INTEGER

```
result = IBSET ( I , POS )
```

Argument(s)

I

I shall be of type INTEGER.

POS

POS shall be of type INTEGER. It shall be non-negative and less than the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* except that bit *POS* is set to one. Note that the lowest order *POS* is zero.

Remarks

IBSET, IIBSET, and JIBSET set a bit *POS* to one in *I*.

The generic name, IBSET, may be used with any INTEGER argument.

The type of the result of each function is the same as the type of the argument.

Example

```
i = ibset (37,1) ! i is assigned the value 39
```

2.221 IBTOD Service Subroutine

Description

The absolute value of *j* is converted to characters and the lower four characters are set to *i*. Leading zeros are changed to blanks.

Syntax

```
CALL  IBTOD ( i , j )
```

Argument(s)

i

Default INTEGER scalar. Variable whose the absolute value is to be set.

j

Default INTEGER scalar. The value to convert.

Example

```
use service_routines,only:ibtod
integer :: i
```

```
call ibtod(i,123456)
write(6,fmt="(1x,a8)") i
end
```

2.222 ICHAR Intrinsic Function

Description

Position of a character in the processor collating sequence associated with the kind of the character.

Class

Elemental function.

Syntax

result = ICHAR (*C* [, *KIND*])

Required Argument(s)

C

C shall be of type CHARACTER and of length one.

Optional Argument(s)

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

The result is of type INTEGER. Its value is the position of *C* in the processor collating sequence associated with the kind of *C* and is in the range 0 <= ICHAR(*C*) <= 255.

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

Example

```
i = ichar('c')  ! i is assigned the value 99 for
                ! character c in the ASCII
                ! collating sequence
```

2.223 IDATE Service Subroutine

Description

Returns the month, day, and year of the current system.

Syntax

CALL IDATE (*ia*)

Argument(s)

ia

Default INTEGER and rank one. Its size shall be at least 3. The values returned in *ia* are as follows:

```
ia(1)     :     Current system day.
ia(2)     :     Current system month.
ia(3)     :     Current system year.
```

Example

```
use service_routines,only:idate
integer :: t(3)
call idate(t)
```

```
write(6,fmt="(1x,i2,'/',i2,'/',i4)") t
end
```

2.224 IEEE_CLASS Intrinsic Module Function

Description

IEEE floating-point number type.

Class

Elemental function.

Syntax

result = IEEE_CLASS(*X*)

Argument(s)

X

X shall be of type REAL.

Result

IEEE_CLASS returns the IEEE floating-point number type.

- If IEEE_SUPPORT_NAN(*X*) is true and if the value of *X* is an exception notification NaN, the result value is IEEE_SIGNALING_NAN.
- If IEEE_SUPPORT_NAN(*X*) is true and if the value of *X* is an exception non-notification NaN, the result value is IEEE_QUIET_NAN.
- If IEEE_SUPPORT_INF(*X*) is true and the value of *X* is negative infinity, the result value is IEEE_NEGATIVE_INF.
- If IEEE_SUPPORT_INF(*X*) is true and the value of *X* is positive infinity, the result value is IEEE_POSITIVE_INF.
- If IEEE_SUPPORT_DENORMAL(*X*) is true and the value of *X* is a negative denormal number, the result value is IEEE_NEGATIVE_DENORMAL.
- If IEEE_SUPPORT_DENORMAL(*X*) is true and the value of *X* is a positive denormal number, the result value is IEEE_POSITIVE_DENORMAL.
- If the value of *X* is a negative normal number, the result value is IEEE_NEGATIVE_NORMAL.
- If the value of *X* is a negative zero, the result value is IEEE_NEGATIVE_ZERO.
- If the value of *X* is a positive zero, the result value is IEEE_POSITIVE_ZERO.
- If the value of *X* is a positive normal number, the result value is IEEE_POSITIVE_NORMAL.
- In other cases, the result value is IEEE_OTHER_VALUE.

The function result type is the derived type TYPE(IEEE_CLASS_TYPE).

If this function is used, the intrinsic module IEEE_ARITHMETIC must be referenced.

When IEEE_SUPPORT_DATATYPE(*X*) is false, this procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
type(ieee_class_type) ::result
real :: x
x = -1.0
if (ieee_support_datatype(x)) then
result = ieee_class(x) ! ieee_negative_normal
end if
if (result .eq. ieee_negative_normal) then
print *, "negative_normal"
end if
```

2.225 IEEE_COPY_SIGN Intrinsic Module Function

Description

Replaces the IEEE sign.

Class

Elemental function.

Syntax

result = IEEE_COPY_SIGN(*X* , *Y*)

Argument(s)

X

X shall be of type REAL.

Y

Y shall be of type REAL.

Result

IEEE_COPY_SIGN returns a value with the sign of *Y* in the *X* absolute value.

The function result type is the same type as *X*.

When IEEE_SUPPORT_DATATYPE(*X*) or IEEE_SUPPORT_DATATYPE(*Y*) is false, this procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real ::result,x,y
x = 2.0
y = -1.0
if (ieee_support_datatype(x) .and. ieee_support_datatype(y)) then
result = ieee_copy_sign(x,y) ! -2.0
end if
```

2.226 IEEE_GET_FLAG Intrinsic Module Subroutine

Description

Gets the IEEE exception flag.

Class

Elemental subroutine.

Syntax

CALL IEEE_GET_FLAG(*FLAG* , *FLAG_VALUE*)

Argument(s)

FLAG

Must be the derived type TYPE(IEEE_FLAG_TYPE). Specify the exception flag that gets information. Must be IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

FLAG_VALUE

Must be a default LOGICAL type, and is the INTENT(OUT) argument. The result value of true is returned if the exception flag specified at *FLAG* is exception notification, and the result value of false is returned in other cases.

Remarks

Returns whether the exception flag is exception notification or not.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::flag_value
call ieee_get_flag(ieee_overflow, flag_value)
```

2.227 IEEE_GET_HALTING_MODE Intrinsic Module Subroutine

Description

Gets the halting mode for IEEE exceptions.

Class

Elemental subroutine.

Syntax

```
CALL IEEE_GET_HALTING_MODE( FLAG , HALTING )
```

Argument(s)

FLAG

Must be the derived type TYPE(IEEE_FLAG_TYPE). Specify the exception flag that gets information. Must be IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

HALTING

Must be a default LOGICAL type, and is the INTENT(OUT) argument. If the exception specified at *FLAG* causes halting, true is returned. Otherwise, false is returned.

Remarks

Returns the halting mode for exceptions.

Example

This example saves the halting mode for IEEE_OVERFLOW. Computation is performed while halting mode execution has been interrupted, and after that the halting mode is restored.

```
use, intrinsic :: ieee_arithmetic
logical ::halting
type(ieee_flag_type) ::flag
flag = ieee_overflow
...
call ieee_get_halting_mode(flag, halting)   !Saves  the halting mode
if(ieee_support_halting(flag)) then
  call ieee_set_halting_mode(flag, .false.) !Interrupts the halting mode
  ...                                       !Computes without halting
end if
call ieee_set_halting_mode(flag, halting)   !Restores the halting mode
```

2.228 IEEE_GET_ROUNDING_MODE Intrinsic Module Subroutine

Description

Gets the IEEE rounding mode.

Class

Subroutine.

Syntax

```
CALL IEEE_GET_ROUNDING_MODE( ROUND_VALUE )
```

Argument(s)

ROUND_VALUE

Must be the derived type TYPE(IEEE_ROUND_TYPE) scalar, and is the INTENT(OUT) argument. The floating-point number rounding mode is returned. If the rounding mode is nearest rounding, IEEE_NEAREST is the value returned. If the rounding mode is truncate, IEEE_TO_ZERO is the value returned. If the rounding mode is rounding up, IEEE_UP is the value returned. If the rounding mode is rounding down, IEEE_DOWN is the value returned. In other cases, IEEE_OTHER is the value returned.

Remarks

Returns the current IEEE rounding mode.

Example

In this example, the rounding mode is saved. The rounding mode is set to rounding to the nearest value (IEEE_NEAREST) and computation is performed, and then the rounding mode is restored.

```
use, intrinsic :: ieee_arithmetic
type(ieee_round_type) ::round_value
real ::x
round_value = ieee_nearest
...
call ieee_get_rounding_mode(round_value) ! Saves the rounding mode
if(ieee_support_datatype(x)) then
  if(ieee_support_rounding(round_value, x)) then
    call ieee_set_rounding_mode(ieee_nearest)
    ...               ! Sets the rounding mode to nearest and computes
  end if
end if
call ieee_set_rounding_mode(round_value) ! Restores the rounding mode
```

2.229 IEEE_GET_STATUS Intrinsic Module Subroutine

Description

Gets the IEEE floating-point status.

Class

Subroutine.

Syntax

`CALL IEEE_GET_STATUS(` *STATUS_VALUE* `)`

Argument(s)

STATUS_VALUE

Must be the derived type TYPE(IEEE_STATUS_TYPE) scalar, and is the INTENT(OUT) argument. Returns the floating-point number status.

Remarks

Returns the current value of the floating-point number status.

The floating-point status is the value of all flags prepared for exceptions, rounding modes, underflow modes, and halting.

Example

In this example, all floating-point statuses are saved. All exception flags are set to exception non-notification, and computation including exception processing is performed. The floating-point statuses are then restored.

```
use, intrinsic :: ieee_arithmetic
type(ieee_status_type) ::status_value
...
call ieee_get_status(status_value) ! Saves the floating-point statuses
call ieee_set_flag(ieee_all, (/.false.,.false.,.false.,.false.,.false./))
```

```
! Sets flags to exception non-notification
... ! Computes, including exception processing
call ieee_set_status(status_value) ! Restores the floating-point statuses
```

2.230 IEEE_GET_UNDERFLOW_MODE Intrinsic Module Subroutine

Description

Gets the IEEE underflow mode.

Class

Subroutine.

Syntax

`CALL IEEE_GET_UNDERFLOW_MODE( GRADUAL )`

Argument(s)

GRADUAL

Must be a default LOGICAL scalar, and is the INTENT(OUT) argument. If the current underflow mode is gradual underflow, true is returned. If the current underflow mode is abrupt underflow, false is returned.

Remarks

Returns the current underflow mode.

Gradual underflow is making an underflow into a denormal number. Underflow is making it zero.

If IEEE_SUPPORT_UNDERFLOW_CONTROL(X) is not true for a particular X, that procedure cannot be called.

Example

In this example, computation is executed in abrupt underflow mode, and then the previous mode is restored.

```
use, intrinsic :: ieee_arithmetic
logical ::save_underflow_mode
real ::x
...
if (ieee_support_underflow_control(x)) then
  call ieee_get_underflow_mode(save_underflow_mode)
  ! Saves the underflow mode
  call ieee_set_underflow_mode(gradual=.false.)
  ... ! Computes in sudden underflow mode
  call ieee_set_underflow_mode(save_underflow_mode)
  ! Restores the underflow mode
end if
```

2.231 IEEE_IS_FINITE Intrinsic Module Function

Description

Determines an IEEE finite number.

Class

Elemental function.

Syntax

`result = IEEE_IS_FINITE( X )`

Argument(s)

X

X shall be of type REAL.

Result

The function result type is a default LOGICAL type.

If *X* is finite, IEEE_IS_FINITE returns true. That is, true is returned if IEEE_CLASS(*X*) is IEEE_NEGATIVE_NORMAL (negative normal number), IEEE_NEGATIVE_DENORMAL (negative denormal number), IEEE_NEGATIVE_ZERO (negative zero), IEEE_POSITIVE_ZERO (positive zero), IEEE_POSITIVE_DENORMAL (positive denormal number), or IEEE_POSITIVE_NORMAL (positive normal number). In other cases, false is returned.

If IEEE_SUPPORT_DATATYPE(*X*) is false, that procedure cannot be called.

Example

Example determined as finite:

```
use, intrinsic :: ieee_arithmetic
logical ::result
real ::x
x = 1.0
if (ieee_support_datatype(x)) then
  result = ieee_is_finite(x) ! True
end if
```

2.232 IEEE_IS_NAN Intrinsic Module Function

Description

Determines an IEEE NaN.

Class

Elemental function.

Syntax

result = IEEE_IS_NAN(*X*)

Argument(s)

X

X shall be of type REAL.

Result

The function result type is a default LOGICAL type.

IEEE_IS_NAN returns true when *X* is IEEE NaN. In other cases, it returns false.

If IEEE_SUPPORT_NAN(*X*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real :: x,y
logical ::result
y = 0.0
x = 0.0/y
if (ieee_support_nan(x)) then
  result = ieee_is_nan(x) ! True
end if
```

2.233 IEEE_IS_NEGATIVE Intrinsic Module Function

Description

Determines an IEEE negative number.

Class

Elemental function.

Syntax

```
result = IEEE_IS_NEGATIVE( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The function result type is a default LOGICAL type.

IEEE_IS_NEGATIVE returns true if the value of *X* is negative. That is, true is returned if IEEE_CLASS(*X*) is IEEE_NEGATIVE_NORMAL (negative normal number), IEEE_NEGATIVE_DENORMAL (negative denormal number), IEEE_NEGATIVE_ZERO (negative zero), or IEEE_NEGATIVE_INF (negative infinity). In other cases, it returns false.

If IEEE_SUPPORT_DATATYPE(*X*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real ::x
logical ::result
x = 0.0
if (ieee_support_datatype(x)) then
  result = ieee_is_negative(x) ! False
end if
```

2.234 IEEE_IS_NORMAL Intrinsic Module Function

Description

Determines an IEEE normal number.

Class

Elemental function.

Syntax

```
result = IEEE_IS_NORMAL( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The function result type is a default LOGICAL type.

IEEE_IS_NORMAL returns true if the value of *X* is normal. That is, true is returned if IEEE_CLASS(*X*) is IEEE_NEGATIVE_NORMAL (negative normal number), IEEE_NEGATIVE_ZERO (negative zero), IEEE_POSITIVE_ZERO (positive zero), or IEEE_POSITIVE_NORMAL (positive normal number). In other cases, it returns false.

If IEEE_SUPPORT_DATATYPE(*X*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real ::x
logical ::result
x = 0.0
if (ieee_support_datatype(x)) then
  result = ieee_is_normal(x) ! True
end if
```

2.235 IEEE_LOGB Intrinsic Module Function

Description

The exponent with the IEEE floating-point number padded part removed.

Class

Elemental function.

Syntax

result = IEEE_LOGB(*X*)

Argument(s)

X

X shall be of type REAL.

Result

The function result type is the same type as *X*.

IEEE_LOGB returns the following results:

- If the value of *X* is 0 and is neither infinity nor NaN, the exponent with the padded part of *X* removed is returned. The value is the same as EXPONENT(*X*)-1.
- If the value of *X* is 0, $-\infty$ is returned when IEEE_SUPPORT_INF(*X*) is true. Otherwise, -HUGE(*X*) is returned and IEEE_DIVIDE_BY_ZERO is issued.
- If the value of *X* is infinity, the result is infinity.
- If the value of *X* is NaN, the result is NaN.

If IEEE_SUPPORT_DATATYPE(*X*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real :: result,x
x = -1.1
if(ieee_support_datatype(x)) then
  result = ieee_logb(x)
end if
! result is 0.0
```

2.236 IEEE_NEXT_AFTER Intrinsic Module Function

Description

Returns the real number value that is closest in the specified direction to the given value.

Class

Elemental function.

Syntax

result = IEEE_NEXT_AFTER(*X* , *Y*)

Argument(s)

X

X shall be of type REAL.

Y

Y shall be of type REAL.

Result

The function result type is the same type as *X*.

IEEE_NEXT_AFTER returns the following results:

- If *X*== *Y*, the result is *X* and an exception is not issued.
- If *X* /= *Y*, the result is the value that can be represented that is near to *X* in the *Y* direction. It is the non-zero value nearest to 0, regardless of the sign. If *X* is finite and IEEE_NEXT_AFTER(*X*, *Y*) is infinite, IEEE_OVERFLOW is issued.
  If IEEE_NEXT_AFTER(*X*, *Y*) is a denormal number, IEEE_UNDERFLOW is issued. In both cases, IEEE_INEXACT is issued.

If IEEE_SUPPORT_DATATYPE(*X*) or IEEE_SUPPORT_DATATYPE(*Y*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real :: result,x,y
x=1.0
y=2.0
if(ieee_support_datatype(x) .and. ieee_support_datatype(y)) then
  result = ieee_next_after(x,y)
end if
! result is 1.0+epsilon(x)
```

2.237 IEEE_REM Intrinsic Module Function

Description

IEEE remainder.

Class

Elemental function.

Syntax

result = IEEE_REM(*X* , *Y*)

Argument(s)

X

X shall be of type REAL.

Y

Y shall be of type REAL.

Result

The function returns a result of the type of the real number with the kind type parameter of the more precise of the two arguments.

IEEE_REM is strictly *X*-*Y**N regardless of the rounding mode. However, N is the integer closest to the precise value of *X* / *Y*.
If | N - *X* / *Y* | =1/2, N must be an even number. If the result is zero, the sign for the zero is the same as that of *X*.

If IEEE_SUPPORT_DATATYPE(*X*) or IEEE_SUPPORT_DATATYPE(*Y*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real :: result,x,y
x = 4.0
y = 3.0
if(ieee_support_datatype(x) .and. ieee_support_datatype(y)) then
  result = ieee_rem(x,y)
end if
! result is 1.0
```

2.238 IEEE_RINT Intrinsic Module Function

Description

Converts to an integer value using the IEEE rounding mode.

Class

Elemental function.

Syntax

result = IEEE_RINT(*X*)

Argument(s)

X

X shall be of type REAL.

Result

The function result type is the same type as *X*.

IEEE_RINT returns the value of *X* rounded to an integer according to the current rounding mode. If the result is zero, the sign for that zero is the same as that of *X*.

If IEEE_SUPPORT_DATATYPE(*X*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real :: result,x
x = 1.1
if (ieee_support_datatype(x)) then
call ieee_set_rounding_mode(ieee_nearest) !Turn nearest the rounding mode.
result = ieee_rint(x)
end if
! result is 1.0
```

2.239 IEEE_SCALB Intrinsic Module Function

Description

Product of the IEEE floating-point number and the integer power of 2.

Class

Elemental function.

Syntax

result = IEEE_SCALB(*X* , *I*)

Argument(s)

X

X shall be of type REAL.

I

I shall be of type INTEGER.

Result

The function result type is the same type as *X*.

IEEE_SCALB returns the following results:

- If the value of *X* multiplied by the *I* power of 2 can be represented as a normal number, the result is that value.
- If *X* is finite and the value of *X* multiplied by the *I* power of 2 is too large, IEEE_OVERFLOW is issued. If true is returned for IEEE_SUPPORT_INF(*X*), the result value is infinite, and has the same sign as *X*. In other cases, the result is SIGN(HUGE(*X*),*X*).
- If the value of *X* multiplied by the *I* power of 2 is too small and precision is lost, IEEE_UNDERFLOW is issued. The result is an absolute value that is nearest to the *I* power of 2 and can be represented, and has the same sign as *X*.
- If *X* is infinite, the result is the same as *X* and an exception is not issued.

If IEEE_SUPPORT_DATATYPE(*X*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real :: result,x
integer ::i
x = 1.0
i = 2
if (ieee_support_datatype(x)) then
  result = ieee_scalb(x,i)
end if
! result is 4.0
```

2.240 IEEE_SELECTED_REAL_KIND Intrinsic Module Function

Description

The kind type parameter value corresponding to the specified precision and exponent range.

Class

Transformational function.

Syntax

result = IEEE_SELECTED_REAL_KIND([*P* , *R*])

Optional Argument(s)

P

P shall be of type INTEGER scalar.

R

R shall be of type INTEGER scalar.

At least one argument must be specified.

Result

The function result type is a default INTEGER type.

IEEE_SELECTED_REAL_KIND returns the value of the kind type parameter of the IEEE real number with a decimal precision of *P* digits or more and a decimal exponent range of *R* or more. If multiple kind type parameters meet the conditions, the value with the

minimum decimal precision is returned. If there is more than one of those, the one with the smallest kind type parameter value is returned. For the data object X of that type, the result of IEEE_SUPPORT_DATATYPE(X) is true.

If the kind type parameter cannot be used, -1 is returned if the precision cannot be used, -2 is returned if the exponent range cannot be used, and -3 is returned if both cannot be used.

Example

```
use, intrinsic :: ieee_arithmetic
integer ::result
result = ieee_selected_real_kind(6,30)
! result is 4
```

2.241 IEEE_SET_FLAG Intrinsic Module Subroutine

Description

Sets the IEEE exception flag.

Class

Pure subroutine.

Syntax

`CALL IEEE_SET_FLAG( FLAG , FLAG_VALUE )`

Argument(s)

FLAG

Must be the derived type TYPE(IEEE_FLAG_TYPE) scalar. If the flag value is IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT, a value is set in the corresponding exception flag. Two entities in *FLAG* cannot have the same value.

FLAG_VALUE

Must be a default LOGICAL type. It must be shape compatible with *FLAG*. If the entity value is true, exception notification is set for the corresponding flag. Otherwise, exception non-notification is set.

Remarks

Either exception notification or exception non-notification is set in the exception flag specified at *FLAG* in accordance with the value specified at *FLAG_VALUE*.

Example

In this example, exception notification is set in the exception flag IEEE_OVERFLOW.

```
use, intrinsic :: ieee_arithmetic
call ieee_set_flag(ieee_overflow, .true.)
```

2.242 IEEE_SET_HALTING_MODE Intrinsic Module Subroutine

Description

Sets the halting mode for IEEE exceptions.

Class

Pure subroutine.

Syntax

`CALL IEEE_SET_HALTING_MODE( FLAG , HALTING )`

Argument(s)

FLAG

Must be the derived type TYPE(IEEE_FLAG_TYPE) scalar. The value must be IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT. Two entities in *FLAG* cannot have the same value.

HALTING

Must be a default LOGICAL scalar. It must be shape compatible with *FLAG*. If the entity value is true, execution is halted if the exception specified at the corresponding *FLAG* entity occurs. Otherwise, execution continues.

Remarks

Controls whether execution continues or halts after an exception occurs.

If IEEE_SUPPORT_HALTING(*FLAG*) is false, that procedure cannot be called.

Example

This example saves the halting mode for IEEE_OVERFLOW, performs computation in the halting mode state that continues execution, and then restores the halting mode.

```
use, intrinsic :: ieee_arithmetic
logical ::halting
type(ieee_flag_type) ::flag
flag = ieee_overflow
...
call ieee_get_halting_mode(flag, halting)  !Saves the halting mode
if (ieee_support_halting(flag)) then
  call ieee_set_halting_mode(flag, .false.)!Sets the halting mode
                                           !to continue execution
  ...                                      !Computes without halting
end if
call ieee_set_halting_mode(flag, halting)  !Restores the halting mode
```

2.243 IEEE_SET_ROUNDING_MODE Intrinsic Module Subroutine

Description

Sets the IEEE rounding mode.

Class

Subroutine.

Syntax

```
CALL IEEE_SET_ROUNDING_MODE( ROUND_VALUE )
```

Argument(s)

ROUND_VALUE

Must be the derived type TYPE(IEEE_ROUND_TYPE) scalar. Set the floating-point number rounding mode.

Remarks

Changes the current IEEE rounding mode to the mode specified at *ROUND_VALUE*.

If IEEE_SUPPORT_ROUNDING(*ROUND_VALUE*, X) is not true for the value of X for which IEEE_SUPPORT_DATATYPE(X) is true, this procedure cannot be called.

Example

In this example, the rounding mode is saved. The rounding mode is set to rounding to the nearest value (IEEE_NEAREST) and computation is performed, and then the rounding mode is restored.

```
use, intrinsic :: ieee_arithmetic
type(ieee_round_type) ::round_value, round_back
```

```
real ::x
round_value = ieee_nearest
...
call ieee_get_rounding_mode(round_back) ! Saves the rounding mode
if(ieee_support_datatype(x)) then
  if(ieee_support_rounding(round_value, x)) then
    call ieee_set_rounding_mode(round_value)
    ... ! Sets the rounding mode to nearest and computes
  end if
end if
call ieee_set_rounding_mode(round_back) ! Restores the rounding mode
```

2.244 IEEE_SET_STATUS Intrinsic Module Subroutine

Description

Sets the IEEE floating-point status.

Class

Subroutine.

Syntax

```
CALL IEEE_SET_STATUS( STATUS_VALUE )
```

Argument(s)

STATUS_VALUE

Must be the derived type TYPE(IEEE_STATUS_TYPE) scalar. Specify the original value of the floating-point status. The value must be the value that was obtained when IEEE_GET_STATUS was called before calling this procedure.

Remarks

Restores the floating point status to the original value.

Example

In this example, all floating-point statuses are saved, all exception flags are set to exception non-notification, and computation including exception processing is performed. After that, the floating-point statuses are restored.

```
use, intrinsic :: ieee_arithmetic
type(ieee_status_type) ::status_value
...
call ieee_get_status(status_value) ! Saves the floating-point statuses
call ieee_set_flag(ieee_all, (/.false.,.false.,.false.,.false.,.false./))
! Sets the flag to exception non-notification
... ! Computes, including exception processing
call ieee_set_status(status_value) ! Restores the floating-point statuses
```

2.245 IEEE_SET_UNDERFLOW_MODE Intrinsic Module Subroutine

Description

Sets the IEEE underflow mode.

Class

Subroutine.

Syntax

```
CALL IEEE_SET_UNDERFLOW_MODE( GRADUAL )
```

Argument(s)

GRADUAL

Must be a default LOGICAL scalar. If the value is true, the current underflow mode is set to gradual underflow. If the value is false, the current underflow mode is set to abrupt underflow.

Remarks

Sets the current underflow mode.

Gradual underflow is making an underflow into a denormal number. Underflow is making it zero.

If IEEE_SUPPORT_UNDERFLOW_CONTROL(X) is false, that procedure cannot be called.

Example

In this example, computation is executed in abrupt underflow mode, and then the previous mode is restored.

```
use, intrinsic :: ieee_arithmetic
logical ::save_underflow_mode
real ::x
...
if (ieee_support_underflow_control(x)) then
  call ieee_get_underflow_mode(save_underflow_mode)
  ! Saves the underflow mode
  call ieee_set_underflow_mode(gradual=.false.)
  ... ! Computes in the abrupt underflow mode
  call ieee_set_underflow_mode(save_underflow_mode)
  ! Restores the underflow mode
end if
```

2.246 IEEE_SUPPORT_DATATYPE Intrinsic Module Function

Description

Determines IEEE arithmetic operation support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_DATATYPE( )        or
result = IEEE_SUPPORT_DATATYPE( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array

Result

The function result type is a default LOGICAL type scalar.

IEEE_SUPPORT_DATATYPE returns true for all REAL types if called with no arguments. It also returns true for variables of REAL types that have the same kind type parameters as the argument, if one is given and if the processor provides IEEE arithmetic. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_datatype()
```

2.247 IEEE_SUPPORT_DENORMAL Intrinsic Module Function

Description

Determines IEEE denormal support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_DENORMAL( )        or
result = IEEE_SUPPORT_DENORMAL( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_DENORMAL(*X*) returns true if the value of IEEE_SUPPORT_DATATYPE(*X*) is true and if the processor provides denormal number arithmetic operations and assignments for REAL types with the same kind type parameters as *X*. In other cases, it returns false.
- IEEE_SUPPORT_DENORMAL() returns true if the value of IEEE_SUPPORT_DENORMAL(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_denormal()
```

2.248 IEEE_SUPPORT_DIVIDE Intrinsic Module Function

Description

Determines IEEE divide support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_DIVIDE( )        or
result = IEEE_SUPPORT_DIVIDE( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_DIVIDE(*X*) returns true if the processor provides division (with the precision prescribed in IEEE international standards) for REAL types with the same kind type parameters as *X*. In other cases, it returns false.
- IEEE_SUPPORT_DIVIDE() returns true if the value of IEEE_SUPPORT_DIVIDE(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_divide()
```

2.249 IEEE_SUPPORT_FLAG Intrinsic Module Function

Description

Determines IEEE exception support.

Class

Inquiry function.

Syntax

result = IEEE_SUPPORT_FLAG(*FLAG*) or
result = IEEE_SUPPORT_FLAG(*FLAG* , *X*)

Required Argument(s)

FLAG

Must be TYPE(IEEE_FLAG_TYPE) scalar. The *FLAG* value must be IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_FLAG(*FLAG*, *X*) returns true if the processor provides the specified exception for REAL types with the same kind type parameters as *X*. In other cases, it returns false.
- IEEE_SUPPORT_FLAG(*FLAG*) returns true if the IEEE_SUPPORT_FLAG(*FLAG*, *X*) value is true for *X* of all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
real ::x
logical ::result
result = ieee_support_flag(ieee_invalid,x)
```

2.250 IEEE_SUPPORT_HALTING Intrinsic Module Function

Description

Determines IEEE halting mode support.

Class

Inquiry function.

Syntax

result = IEEE_SUPPORT_HALTING(*FLAG*)

Argument(s)

FLAG

Must be TYPE(IEEE_FLAG_TYPE) scalar. The *FLAG* value must be IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

Result

The function result type is a default LOGICAL type scalar.

IEEE_SUPPORT_HALTING returns true if the processor can control whether program execution continues or is halted after the exception specified at *FLAG* occurs. In other cases, it returns false.

Example

This example queries whether or not the processor provides the INVALID exception.

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_halting(ieee_invalid)
```

2.251 IEEE_SUPPORT_INF Intrinsic Module Function

Description

Determines IEEE infinity support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_INF( )        or
result = IEEE_SUPPORT_INF( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array.

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_INF(*X*) returns true if the processor provides both positive and negative infinity for REAL types with the same kind type parameters as *X*. In other cases, it returns false.
- IEEE_SUPPORT_INF() returns true if the value of IEEE_SUPPORT_INF(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_inf()
```

2.252 IEEE_SUPPORT_IO Intrinsic Module Function

Description

Determines IEEE format processing support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_IO( )        or
result = IEEE_SUPPORT_IO( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array.

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_IO(*X*) returns true if the processor provides radix conversion for the input/output rounding modes UP, DOWN, ZERO, and NEAREST as prescribed in the IEEE international standards for REAL types with the same kind type parameters as *X* in formatted input/output. In other cases, it returns false.
- IEEE_SUPPORT_IO() returns true if the value of IEEE_SUPPORT_IO(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_io()
```

2.253 IEEE_SUPPORT_NAN Intrinsic Module Function

Description

Determines IEEE NaN support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_NAN( )        or
result = IEEE_SUPPORT_NAN( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array.

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_NAN(*X*) returns true if the processor provides IEEE international standard NaN for REAL types with the same kind type parameters as *X*. In other cases, it returns false.
- IEEE_SUPPORT_NAN() returns true if the value of IEEE_SUPPORT_NAN(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_nan()
```

2.254 IEEE_SUPPORT_ROUNDING Intrinsic Module Function

Description

Determines IEEE rounding mode support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_ROUNDING( ROUND_VALUE )        or
result = IEEE_SUPPORT_ROUNDING( ROUND_VALUE , X )
```

Required Argument(s)

ROUND_VALUE

Must be TYPE(IEEE_ROUND_TYPE).

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array.

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_ROUNDING(*ROUND_VALUE*, *X*) returns true if the processor provides the rounding mode specified at *ROUND_VALUE* for REAL types with the same kind type parameters as *X*. In other cases, it returns false.
- IEEE_SUPPORT_ROUNDING(*ROUND_VALUE*) returns true if the value of IEEE_SUPPORT_ROUNDING(*ROUND_VALUE*,*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
type(ieee_round_type) ::round_value
logical ::result
round_value = ieee_nearest
result = ieee_support_rounding(round_value)
```

2.255 IEEE_SUPPORT_SQRT Intrinsic Module Function

Description

Determines IEEE square root support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_SQRT( )        or
result = IEEE_SUPPORT_SQRT( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array.

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_SQRT(*X*) returns true if the processor implements IEEE international standard square roots for REAL types with the same kind type parameters as *X*. In other cases, it returns false.

- IEEE_SUPPORT_SQRT() returns true if the value of IEEE_SUPPORT_SQRT(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_sqrt()
```

2.256 IEEE_SUPPORT_STANDARD Intrinsic Module Function

Description

Determines IEEE function support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_STANDARD( )        or
result = IEEE_SUPPORT_STANDARD( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array.

Result

The function result type is a default LOGICAL type scalar.

IEEE_SUPPORT_STANDARD determines whether all IEEE functions are provided. The following results are returned:

- IEEE_SUPPORT_STANDARD(*X*) returns true if the result values are all true for the function group 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*). In other cases, it returns false.
- IEEE_SUPPORT_STANDARD() returns true if the value of IEEE_SUPPORT_STANDARD(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_standard()
```

2.257 IEEE_SUPPORT_UNDERFLOW_CONTROL Intrinsic Module Function

Description

Determines IEEE underflow mode support.

Class

Inquiry function.

Syntax

```
result = IEEE_SUPPORT_UNDERFLOW_CONTROL( )        or
result = IEEE_SUPPORT_UNDERFLOW_CONTROL( X )
```

Optional Argument(s)

X

Must be REAL type scalar or a REAL type array.

Result

The function result type is a default LOGICAL type scalar.

- IEEE_SUPPORT_UNDERFLOW_CONTROL(*X*) returns true if the processor provides functions that control the underflow mode for floating-point operations of the same type as *X*. In other cases, it returns false.
- IEEE_SUPPORT_UNDERFLOW_CONTROL() returns true if the value of IEEE_SUPPORT_UNDERFLOW_CONTROL(*X*) is true for *X* for all REAL types. In other cases, it returns false.

Example

```
use, intrinsic :: ieee_arithmetic
logical ::result
result = ieee_support_underflow_control()
```

2.258 IEEE_UNORDERED Intrinsic Module Function

Description

Determines IEEE NaN.

Class

Elemental function.

Syntax

result = IEEE_UNORDERED(*X* , *Y*)

Argument(s)

X

Must be REAL type.

Y

Must be REAL type.

Result

The function result type is a default LOGICAL type scalar.

IEEE_UNORDERED returns true if either or both *X* or *Y* is NaN. In other cases, it returns false.

If IEEE_SUPPORT_DATATYPE(*X*) or IEEE_SUPPORT_DATATYPE(*Y*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real :: x,y
logical :: result
x = 1.0
y = 2.0
if (ieee_support_datatype(x) .and. ieee_support_datatype(y)) then
  result = ieee_unordered(x,y)
end if
```

2.259 IEEE_VALUE Intrinsic Module Function

Description

Generates IEEE floating-point numbers.

Class

Elemental function.

Syntax

```
result = IEEE_VALUE( X , CLASS )
```

Argument(s)

X

Must be REAL type.

CLASS

Must be the derived type TYPE(IEEE_CLASS_TYPE). The following can be specified as the value:

- If IEEE_SUPPORT_NAN(*X*) is true: IEEE_SIGNALING_NAN or IEEE_QUIET_NAN
- If IEEE_SUPPORT_INF(*X*) is true: IEEE_NEGATIVE_INF or IEEE_POSITIVE_INF
- If IEEE_SUPPORT_DENORMAL(*X*) is true: IEEE_NEGATIVE_DENORMAL or IEEE_POSITIVE_DENORMAL
- Values that can be specified unconditionally:
  IEEE_NEGATIVE_NORMAL, IEEE_NEGATIVE_ZERO, IEEE_POSITIVE_ZERO, or IEEE_POSITIVE_NORMAL

Result

The function result type is the same type as *X*.

IEEE_VALUE returns the IEEE real number specified at *CLASS*.

If IEEE_SUPPORT_DATATYPE(*X*) is false, that procedure cannot be called.

Example

```
use, intrinsic :: ieee_arithmetic
real ::result,x
type(ieee_class_type) :: class
class = ieee_positive_normal
if (ieee_support_datatype(x)) then
  result = ieee_value(x,class)
end if
```

2.260 IEOR Intrinsic Function

Description

Bit-wise logical exclusive OR.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
IEOR	---	2	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant	INTEGER
IEOR or XOR	---	2	One-byte INTEGER, One-byte INTEGER	One-byte INTEGER
	IIEOR		Two-byte INTEGER, Two-byte INTEGER	Two-byte INTEGER
	IEOR		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	XOR		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	JIEOR		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER, Eight-byte INTEGER	Eight-byte INTEGER

```
result = IEOR ( I ,  J )
result = XOR ( I ,  J )
```

Argument(s)

I

I shall be of type INTEGER. IEOR may be used a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER and of the same kind as *I*. IEOR may be used a binary, octal or hexadecimal constant.

Result

The result is of type INTEGER. Its value is the value obtained by performing a bit-wise logical exclusive OR of *I* and *J*.

Remarks

IEOR, XOR, IIEOR, and JIEOR evaluate the bit-wise logical exclusive OR of type INTEGER.

The generic names, IEOR and XOR, may be used with any INTEGER argument.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. The binary, octal or hexadecimal constants for both shall not specify.

The type of the result of each function is the same as the type of the argument *I* or *J*.

Example

```
i=53
j=45
k=ieor(i,j)  ! k is assigned the value 24
```

2.261 IERRNO Service Function

Description

Returns the system error code.

Syntax

```
iy = IERRNO ( )
```

Result

Default INTEGER. The system error code.

Example

```
   use service_routines,only:ierrno
   integer :: i
   do i=7,99
     write(unit=i,err=10) i
   end do
10 print *,ierrno()
   end
```

2.262 IETOM Service Subroutine

Description

IEEE-format floating-point data is converted to IBM370-format floating-point data. For information on IEEE-format floating-point data and information on IBM370-format floating-point data, see "Fortran User's Guide".

Syntax

```
CALL  IETOM ( r1, r2, type, retcd )
```

Argument(s)

r1

Default REAL or double precision REAL scalar. IEEE-format floating-point data to convert.

r2

Default REAL or double precision REAL scalar. Returns IBM370-format floating-point data.

type

Default INTEGER scalar. The conversion types as follows:

```
=0    :    Default REAL
=1    :    Double precision REAL
```

retcd

Default INTEGER scalar. Return code as follows:

```
=0    :    Processing ended normally.
=4    :    1 to 3 bits in the mantissa of floating-point data were lost during conversion.
=8    :    Floating-point overflow or underflow, Inf or NaN was detected.
=12   :    The third argument type is invalid.
```

Example

```
use service_routines,only:ietom
real :: ie3
real :: mdata
integer :: ret1,cltype=0
data ie3/11.1e+1/
call ietom(ie3,mdata,cltype,ret1)
end
```

2.263 IF Construct

Description

The IF construct selects for execution no more than one of its constituent blocks.

Syntax

```
[ if-construct-name : ] IF ( scalar-logical-expr ) THEN
    block
[ ELSE IF ( scalar-logical-expr ) THEN [ if-construct-name ]
    block ] ...
[ ELSE [ if-construct-name ]
    block ]
END IF [ if-construct-name ]
```

Where:

construct-name is the name of IF construct.

If the IF THEN statement is identified by an *if-construct-name*, the corresponding END IF statement shall specify the same *if-construct-name*. If the IF THEN statement is not identified by an *if-construct-name*, the corresponding END IF statement shall not

specify an *if-construct-name*. If an ELSE IF statement or ELSE statement is identified by an *if-construct-name*, the corresponding IF THEN statement shall specify the same *if-construct-name*.

scalar-logical-expr is a scalar LOGICAL expression.

block is a sequence of zero or more statements or executable constructs. *block* need not contain any executable statements and constructs. Execution of such a *block* has no effect.

Remarks

At most one of the blocks contained within the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks contained within the construct will be executed. The *scalar-logical-exprs* are evaluated in the order of their appearance in the construct until a true value is found or an ELSE statement or END IF statement is encountered. If a true value or an ELSE statement is found, the block immediately following is executed and this completes the execution of the IF construct. The *scalar-logical-exprs* in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated expressions is true and there is no ELSE statement, the execution of the construct is completed without the execution of any block within the construct.

Example

```
if (a>b) then
  c = d
else if (a<b) then
  d = c
else     ! a==b
  stop
end if
```

2.264 IF Statement

Description

The IF statement controls whether or not a single executable statement is executed.

Syntax

```
IF ( scalar-logical-expr ) action-stmt
```

Where:

scalar-logical-expr is a scalar LOGICAL expression.

action-stmt is an executable statement other than another IF or the END statement.

Remarks

Execution of an IF statement causes evaluation of *scalar-logical-expr*. If the value of *scalar-logical-expr* is true, *action-stmt* is executed. If the value is false, *action-stmt* is not executed.

Example

```
if (a >= b) a = -a
```

2.265 IF THEN Statement

Description

The IF THEN statement begins an IF construct. See "2.263 IF Construct" for IF construct.

Syntax

```
[ if-construct-name : ] IF ( scalar-logical-expr ) THEN
```

Where:

if-construct-name is an optional name for the IF construct.

scalar-logical-expr is a scalar LOGICAL expression.

2.266 IMAGE_INDEX Intrinsic Function

Description

Converts cosubscripts to image index.

Class

Inquiry function.

Syntax

```
result = IMAGE_INDEX ( COARRAY, SUB )
```

Argument(s)

COARRAY

COARRAY must be a coarray.

SUB

SUB must be an integer array of rank one. Its size must be same as the corank of *COARRAY*.

Result

Default integer scalar. If *SUB* is a valid sequence of cosubscripts for *COARRAY*, it is defined with the corresponding image index. Otherwise, it is 0.

Example

```
integer, save :: k[2,*]
if (this_image() == 1) then
  print *,image_index(k,[1,1]) ! 1
  print *,image_index(k,[2,1]) ! 2 if the number of images is more than 1, otherwise 0
  print *,image_index(k,[1,2]) ! 3 if the number of images is more than 2, otherwise 0
  print *,image_index(k,[2,2]) ! 4 if the number of images is more than 3, otherwise 0
end if
```

2.267 IMPLICIT Statement

Description

The IMPLICIT statement specifies, for a scoping unit, a type and optionally a kind or a CHARACTER length for each name beginning with a letter or a '$' specified in the IMPLICIT statement. Alternately, it can specify that no implicit typing is to apply in the scoping unit.

Syntax

```
IMPLICIT implicit-spec-list        or
IMPLICIT NONE
```

Where:

implicit-spec-list is a comma-separated list of

```
type-spec ( letter-spec-list )
```

type-spec is

```
intrinsic-type-spec       or
TYPE ( type-name )        or
CLASS( type-name )        or
CLASS( * )
```

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
UNDEFINED                         or
BYTE
```

kind-selector is

```
( [ KIND = ] kind )          or
* mem-length
```

char-selector is

```
( LEN = char-length-parm , KIND = kind )          or
( char-length-parm , [ KIND = ] kind )            or
( KIND = kind , LEN = char-length-parm )          or
( [ LEN = ] char-length-parm )                    or
* char-length (obsolescent feature)
```

char-length is

```
( char-length-parm )                              or
scalar-int-literal-constant
```

char-length-parm is

```
specification-expr        or
*                         or
:
```

kind is the value of kind type parameter. *kind* is a scalar INTEGER initialization expression.

mem-length is a number of bytes used to represent each respective type. It shall be a scalar INTEGER initialization expression. If the *type-spec* is INTEGER, REAL, or LOGICAL, the value of *mem-length* has the same meaning as the value of *kind*. If the *type-spec* is COMPLEX, the value of *mem-length* has the same meaning as the value of *kind* *2.

See "2.469 Type Declaration Statement" for the value of *kind* and *mem-length*.

A character length parameter value of ':' indicates a deferred type parameter.

type-name is the name of the derived type. *type-name* is defined within the scoping unit that the IMPLICIT statement appeared or is accessible there by use or host association.

letter-spec-list is a comma-separated list of

```
letter [ -letter ]
```

letter is one of the letters A-Z or a '$'.

Remarks

A *letter-spec* consisting of two letters separated by a minus is equivalent to writing a list containing all of the letters in alphabetical order in the alphabetic sequence from the first letter through the second letter. The second letter shall follow the first letter alphabetically. The '$' follows the letter Z in alphabetical order. The same letter shall not appear as a single letter or be included in a range of letters more than once in all of the IMPLICIT statements in a scoping unit.

IMPLICIT NONE specifies the null mapping for all the letters. 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.

The UNDEFINED *type-spec* specifies the null mapping for specified letters.

If a mapping is not specified for a letter, the default for a program unit or an interface body is default INTEGER if the letter is I,J,...,N and default REAL otherwise, and the default for an internal or module procedure is the mapping in the host scoping unit.

Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic function, and is not made accessible by use association or host association is declared implicitly to be of the type (and type parameters, kind and length) mapped from the first letter of its name, provided the mapping is not null.

An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of the result variable of that function subprogram.

See "2.469 Type Declaration Statement" for the CLASS specifier.

Example

```
implicit character (c), integer (a-b, d-z)
                 ! specifies that all data objects
                 ! beginning with c are implicitly of
                 ! type character, and other data
                 ! objects are of type integer
```

2.268 IMPORT Statement

Description

The IMPORT statement specifies that the named entities from the host scoping unit are accessible in the interface body by host association.

Syntax

```
IMPORT [ [ :: ] import-name-list ]
```

Where:

import-name-list is list of import name. the import name must be the name of an entity in the host scoping unit.

Remarks

The IMPORT statement is allowed only in an interface body.

An entity that is imported in the IMPORT statement and, is defined in the host scoping unit must be explicitly declared prior to the interface body.

If an IMPORT statement without *import-name-list* is specified, each host entity not named in an IMPORT statement also is made accessible by host association.

If an IMPORT statement with *import-name-list* is specified, the name of an entity that becomes accessible by the IMPORT statement must not declare as a local variable in the scope.

Example

```
module mod
  integer,parameter::i=1
end module

use mod
interface
  subroutine sub(k)
    import::i            ! i becomes accessible by the IMPORT statement.
    integer(i)::k
  end subroutine
end interface
```

2.269 INCLUDE Line

Description

The INCLUDE line causes text in another file to be processed as if the text therein replaced the INCLUDE line.

Syntax

```
INCLUDE char-const
```

Where:

char-const is a CHARACTER literal constant that enclosed apostrophes or quotation marks. *char-const* is a file name that corresponds to a file that contains source text to be included in place of the INCLUDE line.

Remarks

The INCLUDE line shall be the only non-blank text on this source line other than an optional trailing comment. A statement label or additional statements are not allowed on the line. The INCLUDE line is not a Fortran statement.

Example

```
include "typedef.inc"  ! include a file named typedef.inc
                       ! in place of this INCLUDE line
```

2.270 INDEX Intrinsic Function

Description

Starting position of a substring within a string.

Class

Elemental function.

Syntax

result = INDEX (*STRING* , *SUBSTRING* [, *BACK* , *KIND*])

Required Argument(s)

STRING

STRING shall be of type CHARACTER.

SUBSTRING

SUBSTRING shall be of type CHARACTER with the same kind as *STRING*.

Optional Argument(s)

BACK

BACK shall be of type LOGICAL.

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

If *BACK* is absent or false, the result value is the position number in *STRING* where the first instance of *SUBSTRING* begins or zero if there is no such value or if *STRING* is shorter than *SUBSTRING*. If *SUBSTRING* is of zero length, the result value is one.

If *BACK* is present and true, the result value is the position number in *STRING* where the last instance of *SUBSTRING* begins. If *STRING* is shorter than *SUBSTRING* or if *SUBSTRING* is not in *STRING*, zero is returned. If *SUBSTRING* is of zero length, LEN(*STRING*)+1 is returned.

Example

```
i = index('mississippi', 'si')
                 ! i is assigned the value 4
i = index('mississippi', 'si', back=.true.)
                 ! i is assigned the value 7
```

2.271 INMAX Service Function

Description

Returns the largest number of type integer.

Syntax

```
iy = INMAX ( )
```

Result

Default INTEGER scalar. The largest number of type INTEGER.

Example

```
use service_routines,only:inmax
print *,inmax()   ! Writes 2147483647
end
```

2.272 INQUIRE Statement

Description

The INQUIRE statement inquires about properties of named file or of the connection to a unit.

Syntax

```
INQUIRE ( inquire-spec-list )                        or
INQUIRE ( IOLENGTH = iolength ) output-item-list
```

Where:

inquire-spec-list is a comma-separated list of

```
[ UNIT = ] external-file-unit                  or
FILE = file-name-expr                          or
IOSTAT = io-stat                               or
ERR = err-label                                or
EXIST = exist                                  or
OPENED = opened                                or
NUMBER = number                                or
NAMED = named                                  or
NAME = name                                    or
ACCESS = access                                or
SEQUENTIAL = sequential                        or
DIRECT = direct                                or
FORM = form                                    or
FORMATTED = formatted                          or
UNFORMATTED = unformatted                      or
BINARY = binary                                or
RECL = recl                                    or
NEXTREC = nextrec                              or
BLANK = blank                                  or
POSITION = position                            or
ACTION = action                                or
READ = read                                    or
WRITE = write                                  or
READWRITE = readwrite                          or
DELIM = delim                                  or
PAD = pad                                      or
BLOCKSIZE = blocksize                          or
CARRIAGECONTROL = carriagecontrol              or
CONVERT = convert                              or
FLEN = flen                                    or
ASYNCHRONOUS = asynchronous                    or
DECIMAL = decimal                              or
ENCODING = encoding                            or
ID = id                                        or
IOMSG = iomsg                                  or
PENDING = pending                              or
POS = pos                                      or
```

```
ROUND = round                                   or
SIGN = sign                                     or
SIZE = size                                     or
STREAM = stream
```

Inquire by unit, the INQUIRE statement which uses UNIT= specifier.

Inquire by file, the INQUIRE statement which uses FILE= specifier.

Inquire by output list, the INQUIRE statement uses IOLENGTH= specifier.

An *inquire-spec-list* shall contain one UNIT= specifier or one FILE= specifier, but not both, and at most one of each of the other specifications.

Each specifier cannot be specified more than two times in the *inquire-spec-list*.

If the optional characters UNIT= are omitted from the *external-file-unit* specifier, the *external-file-unit* specifier shall be the first item in the *inquire-spec-list*.

external-file-unit is a scalar INTEGER expression that evaluates to the input/output unit number of an external file.

file-name-expr is a scalar CHARACTER expression that evaluates to the name of a file. Any leading and trailing blanks are ignored. The named file need not exist or be connected to a unit.

io-stat is a scalar INTEGER variable that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, and zero otherwise.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the INQUIRE statement. If an error condition occurs during execution of the INQUIRE statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

exist is a scalar default LOGICAL variable that is assigned the value true if the file specified in the FILE= specifier exists or the input/output unit specified in the UNIT= specifier exists, and false otherwise.

opened is a scalar default LOGICAL variable that is assigned the value true if the file or input/output unit specified is connected, and false otherwise.

number is a scalar INTEGER variable that is assigned the value of the input/output unit of the external file, otherwise the value -1 is assigned.

named is a scalar default LOGICAL variable that is assigned the value true if the file has a name and false otherwise.

name is a scalar default CHARACTER variable that is assigned the name of the file, if the file has a name, otherwise it becomes undefined.

access is a scalar default CHARACTER variable that is assigned the value SEQUENTIAL if the file is connected for sequential access, DIRECT if the file is connected for direct access, STREAM if the file is connected for stream access, or UNDEFINED if the file is not connected.

sequential is a scalar default CHARACTER variable that is assigned the value YES if sequential access is an allowed access method for the file, NO if sequential access is not allowed, and UNKNOWN if the file is not connected.

direct is a scalar default CHARACTER variable that is assigned the value YES if direct access is an allowed access method for the file, NO if direct access is not allowed, and UNKNOWN if the file is not connected.

form is a scalar default CHARACTER variable that is assigned the value FORMATTED if the file is connected for formatted input/output, UNFORMATTED if the file is connected for unformatted input/output, BINARY if the file is connected for binary input/output, and UNDEFINED if there is no connection.

formatted is a scalar default CHARACTER variable that is assigned the value YES if formatted is an allowed form for the file, NO if formatted is not allowed, and UNKNOWN if the file is not connected.

unformatted is a scalar default CHARACTER variable that is assigned the value YES if unformatted is an allowed form for the file, NO if unformatted is not allowed, and UNKNOWN if the file is not connected.

binary is a scalar default CHARACTER variable that is assigned the value YES is binary is an allowed form for the file, NO if binary is not allowed, and UNKNOWN if the file is not connected.

recl is a scalar INTEGER variable that is assigned the value of the record length in bytes for a file connected for direct access, or the maximum record length in bytes for a file connected for sequential access. If the file is not connected, or if connected for stream access, *recl* becomes undefined.

nextrec is a scalar INTEGER variable that is assigned the value *n*+1, where *n* is the number of the last record read or written on the file connected for direct access. If the file has not been written to or read from since becoming connected, the value 1 is assigned. If the file is not connected for direct access or if the file position is unknown, *nextrec* becomes undefined.

blank is a scalar default CHARACTER variable that is assigned the value NULL if null blank control is in effect, ZERO if zero blank control is in effect, and UNDEFINED if the file is not connected for formatted input/output.

position is a scalar default CHARACTER variable that is assigned the value REWIND if the file is connected by an OPEN statement for positioning at its initial point, APPEND if the file is connected for positioning before its endfile record or at its terminal point, ASIS if the file is connected without changing its position. position is after the endfile record, and UNDEFINED if the file is not connected or is connected for direct access.

action is a scalar default CHARACTER variable that is assigned the value READ if the file is connected for input only, WRITE if the file is connected for output only, READWRITE if the file is connected for input and output, and UNDEFINED if the file is not connected.

read is a scalar default CHARACTER variable that is assigned the value YES if READ is an allowed action on the file, NO if READ is not an allowed action of the file, and UNKNOWN if the file is not connected.

write is a scalar default CHARACTER variable that is assigned the value YES if WRITE is an allowed action on the file, NO if WRITE is not an allowed action of the file, and UNKNOWN if the file is not connected.

readwrite is a scalar default CHARACTER variable that is assigned the value YES if READWRITE is an allowed action on the file, NO if READWRITE is not an allowed action of the file, and UNKNOWN if the file is not connected.

delim is a scalar default CHARACTER variable that is assigned the value APOSTROPHE if the apostrophe will be used to delimit character constants written with list-directed or namelist formatting, QUOTE if the quotation mark will be used, NONE if neither quotation marks nor apostrophes will be used, and UNDEFINED if the file is not connected for formatted input/output.

pad is a scalar default CHARACTER variable that is assigned the value YES if the formatted input record is padded with blanks and NO otherwise.

blocksize is a scalar INTEGER variable that is assigned the value of the size, in bytes, of the I/O buffer. This value may be internally adjusted to a record size boundary if the unit has been connected for direct access and therefore may no agree with the BLOCKSIZE= specifier specified in an OPEN statement.

carriagecontrol is a scalar default CHARACTER variable that is assigned the value FORTRAN if the first character of a formatted sequential record is to be used for carriage control, and LIST otherwise.

convert is a scalar default CHARACTER variable that is assigned the value LITTLE_ENDIAN if the file connected for has little endian data, BIG_ENDIAN if the file connected for has big endian data, IBM if the file connected for has IBM-370 format, NATIVE if the file is connected for has native data of the processor, and UNKNOWN if the file is not connected.

flen is a scalar INTEGER variable that is assigned the value of the length of the file in bytes.

iolength is a scalar INTEGER variable that is assigned a value that would result from the use of *output-item-list* in an unformatted output statement. The value can be used as a RECL= specifier in an OPEN statement that connects a file for unformatted direct access when there are input/output statements with the same list of *output-item-list*.

output-item-list is a comma-separated list of

```
expr              or
io-implied-do
```

expr is an expression.

io-implied-do is

```
( output-item-list , io-implied-do-control )
```

io-implied-do-control is

```
do-variable = scalar-expr , scalar-expr [ , scalar-expr ]
```

do-variable is a named scalar variable of type INTEGER, default REAL, or double precision REAL. The *do-variable* of type default REAL or double precision REAL is deleted feature.

scalar-expr is a scalar expression of type INTEGER, default REAL, or double precision REAL. The *scalar-expr* is of type default REAL or double precision REAL is deleted feature.

asynchronous must be a scalar default CHARACTER variable.

The ASYNCHRONOUS= specifier specifies whether an input/output statement is synchronous or asynchronous. 'YES' is assigned to asynchronous if a file is connected and asynchronous input/output is permitted at that device. 'NO' is assigned if a file is connected and asynchronous input/output is not permitted at that device. If a file is not connected, 'UNDEFINED' is assigned to the scalar default CHARACTER variable.

decimal must be a scalar default CHARACTER variable. 'COMMA' or 'POINT' is assigned to *decimal* according to the decimal editing mode enabled at a connection connected as formatted input/output. If not connected, or if connected as something other than formatted input/output, 'UNDEFINED' is assigned to the scalar default CHARACTER variable.

encoding must be a scalar default CHARACTER variable. 'UTF-8' is assigned to *encoding* if the encoding mode of the file connected using formatted input/output is UTF-8. If the file is connected using unformatted input/output, 'UNDEFINED' is assigned. If not connected, and if the processor can determine that the file encoding mode is UTF-8, 'UTF-8' is assigned. If the processor cannot determine the file encoding mode, 'UNKNOWN' is assigned.

id must be a scalar INTEGER expression. The *id* expression value is used at the specified device to identify the unfinished data transfer operation. It is used in conjunction with *pending*.

pending must be a scalar default LOGICAL variable. *pending* is used to judge whether or not a previously unfinished asynchronous data transfer has completed. If *id* is present and if the data transfer operation it specifies has completed, the false value is assigned to the *pending* variable and the INQUIRE statement executes a wait operation for the specified data transfer.

If *id* is not present and if all previously unfinished data transfer operations at that device have been completed, the false value is assigned to the *pending* variable and the INQUIRE statement executes a wait operation for all previously unfinished data transfer operations at that device.

In other cases, the true value is assigned to the *pending* variable and the wait operation is not executed. *pos* must be a scalar default INTEGER variable. The number of the file storage unit immediately after the current position of the file connected as a stream access is assigned to *pos*. If the file is positioned at the end point, the value one greater than the largest file storage unit number in the file is assigned to the variable. If a file is not connected as a stream access, or if the file position is undefined due to an earlier error condition, the variable is undefined.

round must be a scalar default CHARACTER variable. 'UP', 'DOWN', 'ZERO', 'NEAREST', 'COMPATIBLE', or 'PROCESSOR_DEFINED' is assigned to *round* depending on the input/output rounding mode enabled for the connection connected as formatted input/output. If a file is not connected, or if it is not connected as formatted input/output, 'UNDEFINED' is assigned. The processor assigns 'PROCESSOR_DEFINED' only if behavior other than 'UP', 'DOWN', 'ZERO', 'NEAREST', or 'COMPATIBLE' is set as the currently enabled input/output rounding mode.

sign must be a scalar default CHARACTER variable. 'PLUS', 'SUPPRESS', or 'PROCESSOR_DEFINED' is assigned to sign according to the input/output sign mode enabled for the connection connected as formatted input/output. If a file is not connected, or not connected as formatted input/output, 'UNDEFINED' is assigned.

size must be a scalar INTEGER variable. The file size, measured in file storage units, is assigned to *size*. If the file size cannot be determined, -1 is assigned.

stream must be a scalar default CHARACTER variable. 'YES' is assigned to *stream* if the file is connected for stream access. If it is not connected for stream access, 'NO' is assigned. If the processor cannot determine whether it is connected for stream access or not, 'UNKNOWN' is assigned.

iomsg must be a scalar default CHARACTER variable. If the following conditions occur during input/output statement execution, an explanatory message is assigned to *iomsg*:

- Error condition
- File end condition
- Record end condition

In other cases, the *iomsg* value does not change.

Example

```
inquire (unit=8, access=acc, err=200)
  ! what access method for unit 8? go to 200 on error
inquire (this_unit, opened=opnd, direct=dir)
  ! is unit this_unit open? direct access allowed?
```

```
inquire (file="myfile.dat", recl=record_length)
  ! what is the record length of file "myfile.dat"?
```

2.272.1 Output List INQUIRE Statement

The output list INQUIRE statement contains only one IOLENGTH= specifier and output item list.

The number of file storage units required to store the specified output item list data in unformatted files is assigned in the IOLENGTH= specifier scalar INTEGER variable. This value can be specified for the RECL= specifier value in the OPEN statement that connects as an unformatted direct access to files with input/output statements that have the same list as an input item list or an output item list.

2.273 INT Intrinsic Function

Description

Convert to INTEGER type.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
INT1	---	1	INTEGER or REAL or COMPLEX	One-byte INTEGER
INT2	---	1	INTEGER or REAL or COMPLEX	Two-byte INTEGER
	HFIX		Single-prec REAL	Two-byte INTEGER
	IINT		Single-prec REAL	Two-byte INTEGER
	IIFIX		Single-prec REAL	Two-byte INTEGER
	IIDINT		Double-prec REAL	Two-byte INTEGER
INT4	---	1	INTEGER or REAL or COMPLEX	Four-byte INTEGER
JFIX	---	1	INTEGER or REAL or COMPLEX	Four-byte INTEGER
INT	---	1 or 2	INTEGER or REAL or COMPLEX or binary octal, or hexadecimal constant [,INTEGER]	INTEGER
	---	1	One-byte INTEGER	Default INTEGER
	---		Two-byte INTEGER	Default INTEGER
	---		Four-byte INTEGER	Default INTEGER
	---		Eight-byte INTEGER	Default INTEGER
	INT		Single-prec REAL	Default INTEGER
	IFIX		Single-prec REAL	Default INTEGER
	JINT		Single-prec REAL	Default INTEGER
	JIFIX		Single-prec REAL	Default INTEGER
	IDINT		Double-prec REAL	Default INTEGER
	JIDINT		Double-prec REAL	Default INTEGER
	IQINT		Quad-prec REAL	Default INTEGER

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	---		Single-prec COMPLEX	Default INTEGER
	---		Double-prec COMPLEX	Default INTEGER
	---		Quad-prec COMPLEX	Default INTEGER

```
result = INT ( A [ , KIND ] )
```

Required Argument(s)

A

A shall be of type INTEGER, REAL, COMPLEX, or binary, octal, or hexadecimal constant.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

The result's value is the value of *A* without its fractional part. If *A* is of type COMPLEX, the result's value is the value of the real part of *A* without its fractional part. If *A* is a binary, octal, or hexadecimal constant, the result's value is value that is converted integer.

Remarks

INT, INT1, INT2, INT4, HFIX, IINT, IIFIX, IIDINT, IFIX, JINT, JFIX, JIFIX, IDINT, JIDINT, and IQINT convert to integer type.

The generic name, INT1, may be used with any INTEGER, REAL, or COMPLEX argument. The kind type parameter of the result is INTEGER(KIND=1).

The generic name, INT2, may be used with any INTEGER, REAL, or COMPLEX argument. The kind type parameter of the result is INTEGER(KIND=2).

The generic name, INT4, may be used with any INTEGER, REAL, or COMPLEX argument. The kind type parameter of the result is INTEGER(KIND=4).

The generic name, INT, may be used with any INTEGER, REAL, COMPLEX, or binary, octal, or hexadecimal constant argument. The kind type parameter of the result is *KIND* or default INTEGER.

Example

```
i = int(-3.6)  ! i is assigned the value -3
```

2.274 INTEGER Type Declaration Statement

Description

The INTEGER type declaration statement declares entities of type INTEGER. See "2.469 Type Declaration Statement" for type declaration statement.

Syntax

```
INTEGER [ kind-selector ] [ [ , attr-spec ] ... :: ] entity-decl-list
```

2.275 INTENT Statement

Description

The INTENT statement specifies the intended use of a dummy argument.

Syntax

```
INTENT ( intent-spec ) [ :: ] dummy-arg-name-list
```

Where:

intent-spec is

```
IN          or
OUT         or
INOUT
```

dummy-arg-name-list is a comma-separated list of dummy arguments that shall not be a dummy procedure and dummy pointer.

Remarks

An INTENT statement shall appear only in the specification part of a subprogram or an interface body.

The INTENT (IN) attribute for a nonpointer dummy argument specifies that the dummy argument is intended to receive data from the invoking scoping unit. The dummy argument shall not be redefined or become undefined during the execution of the procedure.

A pointer object with the INTENT(IN) attribute must not appear as:

- A pointer object in a nullify statement.
- A data pointer object or procedure pointer object in a pointer assignment statement.
- An allocate object in an allocate statement or deallocate statement.
- 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.

The INTENT (OUT) attribute for a nonpointer dummy argument specifies that the dummy argument is intended to return data to the invoking scoping unit. Any actual argument that becomes associated with such a dummy argument shall be definable except for components of an object of derived type for which default initialization has been specified.

The INTENT(OUT) attribute for a pointer dummy argument specifies that on invocation of the procedure the pointer association status of the dummy argument becomes undefined.

An entity with the INTENT (OUT) attribute must not be of type LOCK_TYPE (1.18.3 LOCK_TYPE Type") or have a subcomponent of this type. An entity with the INTENT (OUT) attribute must not be an allocatable coarray or have a subobject that is an allocatable coarray.

The INTENT (INOUT) attribute specifies that the dummy argument is intended for use both to receive data from and to return data to the invoking scoping unit. Any actual argument that becomes associated with such a dummy argument shall be definable.

If an object has an INTENT attribute, then the subobjects have the same INTENT attribute.

Example

```
subroutine ex (a, b, c)
  real :: a, b, c
  intent (in) a
  intent (out) b
  intent (inout) c
```

2.276 INTERFACE Statement

Description

The INTERFACE statement begins an interface block. An interface block specifies the forms of reference through which a procedure can be invoked. An interface block can be used to specify a procedure interface, a defined operation, or a defined assignment. See "1.12.7 Procedure Interfaces" for interface block.

Syntax

```
INTERFACE [ generic-spec ]        or
ABSTRACT INTERFACE
```

Where:

generic-spec is

```
generic-name                    or
OPERATOR ( defined-operator )   or
ASSIGNMENT ( = )                or
dtio-generic-spec
```

generic-name is the name of a generic procedure.

defined-operator is

```
intrinsic-operator      or
.operator-name.
```

operator-name is a user-defined name for the operation, consisting of one to 240 letters.

ASSIGNMENT(=) is a user-defined assignment.

dtio-generic-spec is derived type input/output procedure that is the following syntax:

```
READ ( FORMATTED )          or
READ ( UNFORMATTED )        or
WRITE ( FORMATTED )         or
WRITE ( UNFORMATTED )
```

Remarks

A generic name in an INTERFACE statement specifies a single name to reference all if the procedure names in the interface block. A generic name may be the same as any one of the procedure names in the interface block, or the same as any accessible generic name.

If OPERATOR is specified, all of the procedures specified in the interface block shall be functions. In the case of functions of two arguments, infix binary operator notation is implied. In the case of functions of one argument, prefix operator notation is implied. The dummy arguments shall be nonoptional dummy data objects and shall be specified with INTENT(IN). A defined operation is treated as a reference to the function. For a unary defined operation, the operand corresponds to the function's dummy argument; for a binary operation, the left-hand operand corresponds to the first argument of the function and the right-hand operand corresponds to the second dummy argument. A given defined operator may, as with generic names, apply to more than one function, in which case it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Because both forms of each relational operator have the same interpretation, extending one form (such as <=) has the effect of defining both forms (<= and .LE.).

If ASSIGNMENT is specified, all of the procedures specified in that interface block shall be subroutines. Each of these subroutines shall have exactly two dummy arguments. Each argument shall be nonoptional. The first argument shall have INTENT(OUT) or INTENT(INOUT) and the second argument shall have INTENT(IN). A defined assignment is treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parenthesis as the second argument. The ASSIGNMENT generic specification specifies that the assignment operation is extended.

However, if both equal signs are the same derived type, the assignment operation is redefined.

The interface block from ABSTRACT INTERFACE is an abstract interface block.

In the ABSTRACT INTERFACE, the FUNCTION statement function name and the SUBROUTINE statement subroutine name cannot be the same as the intrinsic type.

Example

```
interface                       ! interface without generic specification
  subroutine ex (a, b, c)
    implicit none
    real, dimension (2,8) :: a, b, c
    intent (in) a
    intent (out) b
  end subroutine ex
  function why (t, f)
    implicit none
    logical, intent (in) :: t, f
    logical :: why
```

```
  end function why
end interface
interface swap                  ! generic swap routine
  subroutine real_swap(x, y)
    implicit none
    real, intent (inout) :: x, y
  end subroutine real_swap
  subroutine int_swap(x, y)
    implicit none
    integer, intent (inout) :: x, y
  end subroutine int_swap
end interface swap
interface operator (*)          ! use * for set intersection
  function set_intersection (a, b)
    use set_module              ! contains definition of type set
    implicit none
    type (set), intent (in) :: a, b
    type (set) :: set_intersection
  end function set_intersection
end interface operator(*)
interface assignment (=)        ! use = for integer to bit
  subroutine integer_to_bit (n, b)
    implicit none
    integer, intent (out) :: n
    logical, intent (in) :: b(:)
  end subroutine integer_to_bit
end interface assignment(=)
```

2.277 INTRINSIC Statement

Description

The INTRINSIC statement specifies a list of names that represent intrinsic procedures. Doing so permits a name that represents a specific intrinsic function to be used as an actual argument.

Syntax

`INTRINSIC [ :: ]` *`intrinsic-procedure-name-list`*

Where:

intrinsic-procedure-name-list is a comma-separated list of intrinsic procedures.

Remarks

The appearance of a generic intrinsic function name in an INTRINSIC statement does not cause that name to lose its generic property.

If the specific name of an intrinsic function is used as an actual argument, the name shall either appear in an INTRINSIC statement or be given the intrinsic attribute in a type declaration statement in the scoping unit.

Example

```
intrinsic :: dlog, dabs  ! dlog and dabs allowed as
                         ! actual arguments
call zee (a, b, dlog)
```

2.278 IOINIT Service Function

Description

Set the following information of input/output.

CARRIAGECONTROL
PAD
POSITION

These information are in effective when the first input/output statement is executed however the OPEN statement is not executed.

Syntax

```
ly = IOINIT ( cctl , blank , position , prefix , info )
```

Argument(s)

cctl

Default LOGICAL scalar. Same effect as CARRIAGECONTROL= specifier in OPEN statement. If .TRUE., the character in first column of formatted sequential record are treated as follows:

```
blank  : Advance to the next line and print the line.
+      : Do not advance to the next line before printing the line (over-print the current line).
0      : Advance two lines and print the line.
1      : Advance to the first line of the next page and print the line.
```

blank

Default LOGICAL scalar. Same effect as BLANK= specifier in OPEN statement. If .TRUE., blanks in numeric format in input record are treated zero.

position

Default LOGICAL scalar. Same effect as POSITION='APPEND' in OPEN statement. If .TRUE., on sequential access, the current position assigned previously end-of-file record if it exists.

prefix

Default CHARACTER scalar. Specify the environment variable name of previous connected file name. Same effect as environment variable.

info

Default LOGICAL scalar. If .TRUE., each argument value are output.

Result

Default LOGICAL scalar. .TRUE. if successful, .FALSE. if error occurred.

Remarks

Specified values by IOINIT service function are ignored for following files:

- standard error file

Specified values by IOINIT service function are in effective if the first input/output statement is executed for correspondence unit. For example, in the following program, the (1) IOINIT service function call are in effective, however the (2) IOINIT service function call is not in effective.

```
use service_routines,only:ctime,time,ioinit
logical :: l
l = ioinit ( .true., .true., .true., 'FORT', .true.)           ! (1)
write(10,fmt='(2a)')' Fortran:',ctime(time())
l = ioinit ( .false., .true., .true., 'FORT', .true.)          ! (2)
write(10,fmt='(2a)')' Fortran:',ctime(time())
close(10)
end
```

Example

```
use service_routines,only:ctime,time,ioinit
logical::l
open(11,file='a.file',status='replace')
write(11,*) 'Fortran:',ctime(time())
close(11)
l=ioinit ( .true., .true., .true., 'FORT', .true.)
write(10,fmt='(2a)')' Fortran:',ctime(time())
```

```
close(10)
end
```

2.279 IOR Intrinsic Function

Description

Bit-wise logical inclusive OR.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
IOR	---	2	INTEGER, or binary, octal or hexadecimal constant , INTEGER, or binary, octal or hexadecimal constant	INTEGER
IOR or OR	---	2	One-byte INTEGER, One-byte INTEGER	One-byte INTEGER
	IIOR		Two-byte INTEGER, Two-byte INTEGER	Two-byte INTEGER
	IOR		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	OR		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	JIOR		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER, Eight-byte INTEGER	Eight-byte INTEGER

```
result = IOR ( I , J )
result = OR ( I , J )
```

Argument(s)

I

I shall be of type INTEGER. IOR may be used a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER and of the same kind as *I*. IOR may be used a binary, octal or hexadecimal constant.

Result

The result is of type INTEGER and of the same kind as *I*.

Remarks

IOR, OR, IIOR, and JIOR evaluate the bit-wise logical inclusive OR of type INTEGER.

The generic names, IOR and OR, may be used with any INTEGER argument.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. The binary, octal or hexadecimal constants for both shall not specify.

The type of the result of each function is the same as the type of the argument *I* and *J*.

Example

```
i=53
j=45
k=ior(i,j)  ! k is assigned the value 61
```

2.280 IOSTAT_MSG Service Subroutine

Description

IOSTAT_MSG gets a runtime error message.

Syntax

```
CALL  IOSTAT_MSG ( errnum , message )
```

Argument(s)

errnum

errnum shall be of type default INTEGER scalar and has INTENT(IN) attribute. It is an error number of runtime.

message

message shall be of type default CHARACTER scalar and has INTENT(OUT) attribute. It is assigned the runtime error message corresponding to *errnum*.

Example

```
use service_routines,only:iostat_msg
character(len=150) :: msg
call iostat_msg(111,msg)
write (*,*) msg
end
```

2.281 IRAND Service Function

Description

Generates a random number between 0 and 2147483647.

Syntax

```
iy = IRAND ( i )
```

Argument(s)

i

Default INTEGER scalar.

Result

Default INTEGER scalar. The values are as follows:

The value of *i*	Selection process
0	The next random number in the sequence is selected.
1	The generator is restarted and the first random value is selected.
Otherwise	The generator is reseeded using *i*, restarted, and the first random value is selected.

Example

```
use service_routines,only:irand
do i=1,10
  print *,irand(0)    ! Generates 10 random numbers
end do
end
```

2.282 ISATTY Service Function

Description

Returns 1 if a unit is associated with a terminal device, 0 otherwise.

Syntax

```
ly = ISATTY ( unit )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a file.

Result

Default LOGICAL scalar. Returns .TRUE. if *unit* is associated with a terminal device, .FALSE. otherwise.

Example

```
use service_routines,only:isatty
print *,isatty(5)                ! Output: T
print *,isatty(10)               ! Output: F
end
```

2.283 ISHA Intrinsic Function

Description

Arithmetic shift.

Class

Elemental function.

Syntax

```
result = ISHA ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. Its absolute value shall be less than or equal to the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* arithmetic shifted by *SHIFT* positions; if *SHIFT* is positive, the shift is to the left, if *SHIFT* is negative, the shift is to the right.

Example

```
i = isha(223606,5)   ! i is assigned the value 7155392
i = isha(-14142,-5)  ! i is assigned the value -442
```

2.284 ISHC Intrinsic Function

Description

Circular shift.

Class

Elemental function.

Syntax

```
result = ISHC ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. Its absolute value shall be less than or equal to the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* circularly shifted by *SHIFT* positions; if *SHIFT* is positive, the shift is to the left, if *SHIFT* is negative, the shift is to the right.

Example

```
i = ishc(223606,5)   ! i is assigned the value 7155392
i = ishc(-14142,-5)  ! i is assigned the value 402652742
```

2.285 ISHFT Intrinsic Function

Description

Bit-wise shift.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ISHFT	---	2	One-byte INTEGER , INTEGER	One-byte INTEGER
	IISHFT		Two-byte INTEGER , INTEGER	Two-byte INTEGER
	ISHFT		Four-byte INTEGER , INTEGER	Four-byte INTEGER
	JISHFT		Four-byte INTEGER , INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER , INTEGER	Eight-byte INTEGER

```
result = ISHFT ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. Its absolute value shall be less than or equal to the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* shifted by *SHIFT* positions; if *SHIFT* is positive, the shift is to the left, if *SHIFT* is negative, the shift is to the right. Bits shifted out are lost.

Remarks

ISHFT, IISHFT, and JISHFT evaluate the bit-wise logical shift of type INTEGER.

The generic name, ISHFT, may be used with any INTEGER argument.

The type of the result of each function is the same as the type of the argument.

Example

```
i = ishft(3,2)  ! i is assigned the value 12
```

2.286 ISHFTC Intrinsic Function

Description

Bit-wise circular shift of rightmost bits.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
ISHFTC	---	2 or 3	One-byte INTEGER , INTEGER [,INTEGER]	One-byte INTEGER
	IISHFTC		Two-byte INTEGER , INTEGER [,INTEGER]	Two-byte INTEGER
	ISHFTC		Four-byte INTEGER , INTEGER [,INTEGER]	Four-byte INTEGER
	JISHFTC		Four-byte INTEGER , INTEGER [,INTEGER]	Four-byte INTEGER
	---		Eight-byte INTEGER , INTEGER [,INTEGER]	Eight-byte INTEGER

result = ISHFTC (*I* , *SHIFT* [, *SIZE*])

Required Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. The absolute value of *SHIFT* shall be less than or equal to *SIZE*.

Optional Argument(s)

SIZE

SIZE shall be of type INTEGER. The value of *SIZE* shall be positive and shall not be greater than BIT_SIZE (*I*). If absent, it is as if *SIZE* were present with the value BIT_SIZE (*I*).

Result

The result is of type INTEGER and of the same kind as *I*. Its value is equal to the value of *I* with its rightmost *SIZE* bits circularly shifted left by *SHIFT* positions.

Remarks

ISHFTC, IISHFTC, and JISHFTC evaluate the bit-wise circular shift of rightmost bits of type INTEGER.

The generic name, ISHFTC, may be used with any INTEGER argument.

The type of the result of each function is the same as the type of the argument.

Example

```
i = ishftc(13,-2,3)  ! i is assigned the value 11
```

2.287 ISHL Intrinsic Function

Description

Logical shift.

Class

Elemental function.

Syntax

```
result = ISHL ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. Its absolute value shall be less than or equal to the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* logical shifted by *SHIFT* positions; if *SHIFT* is positive, the shift is to the left, if *SHIFT* is negative, the shift is to the right.

Example

```
i = ishl(223606,5)   ! i is assigned the value 7155392
i = ishl(-14142,-5)  ! i is assigned the value 134217286
```

2.288 ISO_C_BINDING Intrinsic Module

The following specifications are offered by intrinsic module ISO_C_BINDING.

- Named constants for interoperable with C program
- Types for interoperable with C program
- Intrinsic procedures for interoperable with C program

See "Fortran User's Guide" for named constants and types for interoperable with C program.

See "2.100 C_ASSOCIATED Intrinsic Module Function", "2.101 C_FUNLOC Intrinsic Module Function", "2.102 C_F_POINTER Intrinsic Module Subroutine", "2.103 C_F_PROCPOINTER Intrinsic Module Subroutine", "2.104 C_LOC Intrinsic Module Function", and "2.105 C_SIZEOF Intrinsic Module Function" for "Intrinsic procedures for interoperable with C program".

2.289 ISO_FORTRAN_ENV Intrinsic Module

Description

This standard intrinsic module enables the reference to named constant, derived type, and procedure.

The module enables the reference to the following named constants:

Named constant	Represents	Type name	Value
CHARACTER_STORAGE_SIZE	The number of bit of character storage unit	Four-byte INTEGER	8
ERROR_UNIT	The number of external file unit for output some diagnostic messages during execution	Four-byte INTEGER	0
FILE_STORAGE_SIZE	The number of bit of file storage unit	Four-byte INTEGER	8

Named constant	Represents	Type name	Value
INPUT_UNIT	The number of external file unit for READ(UNIT=*)	Four-byte INTEGER	5
IOSTAT_END	The value of IOSTAT= specifier, if an end of file condition occurs during execution	Four-byte INTEGER	-1
IOSTAT_EOR	The value of IOSTAT= specifier, if an end of record condition occurs during execution	Four-byte INTEGER	-2
NUMERIC_STORAGE_SIZE	The number of bit of numeric storage unit	Four-byte INTEGER	32
OUTPUT_UNIT	The number of external file unit for WRITE(UNIT=*)	Four-byte INTEGER	6
INT8	Kind type parameter of integer type that occupies 8 bits	Four-byte INTEGER	1
INT16	Kind type parameter of integer type that occupies 16 bits	Four-byte INTEGER	2
INT32	Kind type parameter of integer type that occupies 32 bits	Four-byte INTEGER	4
INT64	Kind type parameter of integer type that occupies 64 bits	Four-byte INTEGER	8
REAL32	Kind type parameter of real type that occupies 32 bits	Four-byte INTEGER	4
REAL64	Kind type parameter of real type that occupies 64 bits	Four-byte INTEGER	8
REAL128	Kind type parameter of real type that occupies 128 bits	Four-byte INTEGER	16
CHARACTER_KINDS	Value of kind type parameter supported as character type	Four-byte INTEGER	1
INTEGER_KINDS	Value of kind type parameter supported as integer type	Four-byte INTEGER	(/1,2,4,8/)
LOGICAL_KINDS	Value of kind type parameter supported as logical type	Four-byte INTEGER	(/1,2,4,8/)
REAL_KINDS	Value of kind type parameter supported as real type	Four-byte INTEGER	(/4,8,16/)
ATOMIC_INT_KIND	Value of kind type parameter of integer variables for which atomic subroutines is supported	Four-byte INTEGER	4
ATOMIC_LOGICAL_KIND	Value of kind type parameter of logical variables for which atomic subroutines is supported	Four-byte INTEGER	4
STAT_LOCKED	The value of STAT= specifier, if a lock variable is locked by executing image	Four-byte INTEGER	1732
STAT_LOCKED_OTHER_IMAGE	The value of STAT= specifier, if a lock variable is locked by another image	Four-byte INTEGER	1733
STAT_STOPPED_IMAGE	The value of STAT= specifier, if a synchronization with an image that has initiated termination of execution is	Four-byte INTEGER	1731

Named constant	Represents	Type name	Value
	required to execute the statement with that specifier		
STAT_UNLOCKED	The value of STAT= specifier, if a lock variable is unlocked	Four-byte INTEGER	1734

The module enables the reference of COMPILER_OPTIONS (see "2.76 COMPILER_OPTIONS Intrinsic Module Function") and COMPILER_VERSION (see "2.77 COMPILER_VERSION Intrinsic Module Function") intrinsic module functions.

The module enables the reference of LOCK_TYPE derived type ("1.18.3 LOCK_TYPE Type") for lock variables.

Example

```
use, intrinsic :: iso_fortran_env, only : input_unit
integer :: input
read ( input_unit, * ) input
```

2.290 IS_CONTIGUOUS Intrinsic Function

Description

Indicate whether contiguity of an array (see "2.82 CONTIGUOUS Statement").

Class

Inquiry function.

Syntax

```
result = IS_CONTIGUOUS (ARRAY )
```

Argument(s)

ARRAY

ARRAY shall be an array. It shall not be a pointer that is disassociated. It shall not be a zero-size or zero-length array.

Result

The result is a scalar of default LOGICAL type. It has the value true if *ARRAY* is contiguous and false if *ARRAY* is not contiguous.

Example

```
integer,target,dimension(5)::target
integer,pointer,dimension(:)::ptr
logical::flag
ptr=>target
flag = is_contiguous(ptr)   ! flag is assigned the value true
ptr=>target(1:5:2)
flag = is_contiguous(ptr)   ! flag is assigned the value false
```

2.291 IS_IOSTAT_END Intrinsic Function

Description

The IS_IOSTAT_END intrinsic function returns whether or not the value stamps the file end condition.

Class

Elemental function.

Syntax

```
result = IS_IOSTAT_END ( I )
```

Argument(s)

I

I shall be of type INTEGER.

Result

The result is of type default LOGICAL.

If *I* is the IOSTAT= specifier scalar INTEGER variable value that indicates the file end condition, true is returned. If not the same, false is returned.

2.292 IS_IOSTAT_EOR Intrinsic Function

Description

The IS_IOSTAT_EOR intrinsic function returns whether or not the value stamps the record end condition.

Class

Elemental function.

Syntax

```
result = IS_IOSTAT_EOR ( I )
```

Argument(s)

I

I shall be of type INTEGER.

Result

The result is of type default LOGICAL.

If *I* is the IOSTAT= specifier scalar INTEGER variable value that indicates the record end condition, true is returned. If not the same, false is returned.

2.293 ITIME Service Subroutine

Description

Returns hours, minutes, and seconds.

Syntax

```
CALL  ITIME ( ia )
```

Argument(s)

ia

Default INTEGER array of rank one. Its size shall be at least 3.
The values returned in *ia* are as follows:

```
ia(1) : Hours.
ia(2) : Minutes.
ia(3) : Seconds.
```

Example

```
use service_routines,only:itime
integer :: t(3)
call itime(t)
write(6,fmt="(1x,i2,':',i2,':',i2)") t
end
```

2.294 IVALUE Service Subroutine

Description

The value of the third argument, unchanged, is transferred to the first argument. The length is four bytes multiplied by the value of second argument.

Syntax

```
CALL  IVALUE ( i , j , k )
```

Argument(s)

i

Default INTEGER scalar. Variable that receives the value of *k*.

j

Default INTEGER scalar. The number to move.

k

Default INTEGER scalar. The value to move.

Example

```
use service_routines,only:ivalue
integer :: i(3),k
open(10,file='x.dat')
write(10,*)1
rewind(10)
read(10,'(a)')k
call ivalue(i,3,k)
end
```

2.295 IZEXT Intrinsic Function

Description

Zero extension.

Class

Elemental function.

Syntax

```
result = IZEXT ( A )
result = IZEXT2 ( A )
result = JZEXT ( A )
result = JZEXT2 ( A )
result = JZEXT4 ( A )
```

Argument(s)

A

For IZEXT, *A* shall be of type one-byte LOGICAL.

For IZEXT2, *A* shall be of type two-byte INTEGER.

For JZEXT, *A* shall be of type four-byte LOGICAL.

For JZEXT2, *A* shall be of type two-byte INTEGER.

For JZEXT4, *A* shall be of type four-byte INTEGER.

Result

The result is of type INTEGER. Its value is *A* treated as unsigned value extended with zeros.

For IZEXT, the result is of type two-byte INTEGER.

For IZEXT2, the result is of type two-byte INTEGER.

For JZEXT, the result is of type four-byte INTEGER.

For JZEXT2, the result is of type four-byte INTEGER.

For JZEXT4, the result is of type four-byte INTEGER.

Example

```
i = jzext2(-4_2)   ! i is assigned the value 65532
```

2.296 JDATE Service Function

Description

```
Returns an 8-character string in the form "yyddd   ".
```

Syntax

```
ch = JDATE ( )
```

Result

8 byte default CHARACTER scalar. The string is a five-digit number whose first two digits are the last two digits of the year and whose final three represent the day of the year (1 for January 1, 366 for December 31 of a leap year, and so on), and three spaces.

Example

```
use service_routines,only:jdate
character(len=8) :: dt
dt = jdate()
end
```

2.297 KILL Service Function

Description

Sends a signal to a user's or group's process.

Syntax

```
iy = KILL ( pid , sig )
```

Argument(s)

pid

Default INTEGER scalar. ID of a process to be signaled.

sig

Default INTEGER scalar. Signal value.

Result

Default INTEGER scalar. Zero if the call was successful; otherwise, a system error code.

Example

```
use service_routines,only:kill,getpid
i = kill(getpid(),9)
end
```

2.298 KIND Intrinsic Function

Description

Kind type parameter.

Class

Inquiry function.

Syntax

```
result = KIND ( X )
```

Argument(s)

X

X can be of any intrinsic type.

Result

The result is a default INTEGER scalar. Its value is equal to the kind type parameter value of *X*.

Example

```
i = kind (0.0) ! i is assigned the value 4
```

2.299 LBOUND Intrinsic Function

Description

Lower bounds of an array or a dimension of an array.

Class

Inquiry function.

Syntax

```
result = LBOUND ( ARRAY [ , DIM , KIND ] )
```

Required Argument(s)

ARRAY

ARRAY can be of any type. It shall not be a scalar and shall not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Argument(s)

DIM

DIM shall be of type INTEGER and shall be a dimension of *ARRAY*.

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

If *DIM* is specified, it is scalar. If omitted, it is one-dimensional array of size n. The n is a rank of *ARRAY*.

If *DIM* is present, the result is the value of the lower bound of *DIM*. If *DIM* is absent, the result is an array of rank one with values corresponding to the lower bounds of each dimension of A*RRAY*.

The lower bound of an array section is always one. The lower bound of zero-sized dimension is also always one.

Example

```
integer, dimension (3,-4:0) :: i
integer :: k,j(2)
j = lbound (i)     ! j is assigned the value [1 -4]
k = lbound (i, 2)  ! k is assigned the value -4
```

2.300 LCOBOUND Intrinsic Function

Description

Returns lower cobound(s) of a coarray.

Class

Inquiry function.

Syntax

```
result = LCOBOUND ( COARRAY [, DIM, KIND ] )
```

Required Argument(s)

COARRAY

COARRAY must be a coarray. If it is allocatable, it must be allocated.

Optional Argument(s)

DIM

DIM must be an integer scalar with a value in the range of 1 and n, where n is the corank of *COARRAY*. The corresponding actual argument must not be an optional dummy argument.

KIND

KIND must be a scalar integer initialization expression.

Result

It is of type integer. If *KIND* is specified, its kind type parameter is the value of *KIND*. Otherwise, the kind type parameter is default integer. If *DIM* is specified, *result* is defined with the value of lower cobound for cosubscript *DIM* of *COARRAY*. Otherwise, it is an array of rank one and size n, where n is the corank of *COARRAY*. In that case, the value of each element of *result* is each lower cobound of the corresponding cosubscript.

Example

```
integer, save :: k[2,3:4,*]
if (this_image()==1) then
  print *,lcobound(k,dim=1) ! 1 is output
  print *,lcobound(k,dim=2) ! 3 is output
  print *,lcobound(k,dim=3) ! 1 is output
end if
```

2.301 LEADZ Intrinsic Function

Description

Number of leading zero bits.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
LEADZ	---	1	INTEGER	Default INTEGER

```
result = LEADZ ( I )
```

Argument(s)

I

I shall be of type INTEGER.

Result

The result is of the default INTEGER type.

Remarks

LEADZ evaluates number of leading zero bits in *I*. If the value of *I* is zero, the result has the value BIT_SIZE(*I*).

The result is of the default INTEGER type.

Example

```
k = leadz(-1)    ! k is assigned the value 0
m = leadz(0)     ! m is assigned the value 32
n = leadz(1)     ! n is assigned the value 31
```

2.302 LEN Intrinsic Function

Description

Length of a CHARACTER data object.

Class

Inquiry function.

Syntax

```
result = LEN ( STRING [ , KIND ] )
```

Required Argument(s)

STRING

STRING shall be of type CHARACTER. It must be scalar or an array. If it is either a disassociated pointer or an unallocated allocatable object, the character length parameter cannot be deferred.

Optional Argument(s)

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

Its value is the number of characters in *STRING* or in an element of *STRING* if *STRING* is array-valued.

If *STRING* is scalar, the number of characters in *STRING* is returned.

If *STRING* is an array, the number of characters in the *STRING* entity is returned.

Example

```
i = len ('Fujitsu') ! i is assigned the value 7
```

2.303 LEN_TRIM Intrinsic Function

Description

Length of a CHARACTER entity without trailing blanks.

Class

Elemental function.

Syntax

```
result = LEN_TRIM ( STRING [ , KIND ] )
```

Required Argument(s)

STRING

STRING shall be of type CHARACTER. It can be scalar or array-valued.

Optional Argument(s)

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

Its value is the number of characters in *STRING* (or in an element of *STRING* if *STRING* is array-valued) minus the number of trailing blanks.

Example

```
i = len_trim ('Fujitsu  ')   ! i is assigned the value 7
i = len_trim ('   ')         ! i is assigned the value zero
```

2.304 LGE Intrinsic Function

Description

Test whether a string is lexically greater than or equal to another string based on the ASCII collating sequence.

Class

Elemental function.

Syntax

```
result = LGE ( STRING_A , STRING_B )
```

Argument(s)

STRING_A

STRING_A shall be of type default CHARACTER.

STRING_B

STRING_B shall be of type default CHARACTER.

Result

The result is of type default LOGICAL. Its value is true if *STRING_B* precedes *STRING_A* in the ASCII collating sequence, or if the strings are the same ignoring trailing blanks; otherwise the result is false. If both strings are of zero length the result is true.

Example

```
logical l
l = lge('elephant', 'horse') ! l is assigned the value false
```

2.305 LGT Intrinsic Function

Description

Test whether a string is lexically greater than another string based on the ASCII collating sequence.

Class

Elemental function.

Syntax

```
result = LGT ( STRING_A , STRING_B )
```

Argument(s)

STRING_A

STRING_A shall be of type default CHARACTER.

STRING_B

STRING_B shall be of type default CHARACTER.

Result

The result is of type default LOGICAL. Its value is true if *STRING_B* precedes *STRING_A* in the ASCII collating sequence; otherwise the result is false. If both strings are of zero length the result is false.

Example

```
logical l
l = lgt('elephant', 'horse') ! l is assigned the value false
```

2.306 LINK Service Function

Description

Makes a link to an existing file.

Syntax

```
iy = LINK ( path1 , path2 )
```

Argument(s)

path1

Default CHARACTER scalar. A pathname of an existing file.

path2

Default CHARACTER scalar. A pathname to be linked to file *path1*. *path2* shall not already exist.

Result

Zero if the call was successful; otherwise, a system error code.

Example

```
use service_routines,only:link
i = link('libx.a','../libp.a')
end
```

2.307 LLE Intrinsic Function

Description

Test whether a string is lexically less than or equal to another string based on the ASCII collating sequence.

Class

Elemental function.

Syntax

```
result = LLE ( STRING_A , STRING_B )
```

Argument(s)

STRING_A

STRING_A shall be of type default CHARACTER.

STRING_B

STRING_B shall be of type default CHARACTER.

Result

The result is of type default LOGICAL. Its value is true if *STRING_A* precedes *STRING_B* in the ASCII collating sequence, or if the strings are the same ignoring trailing blanks; otherwise the result is false. If both strings are of zero length the result is true.

Example

```
logical l
l = lle('elephant', 'horse') ! l is assigned the value true
```

2.308 LLT Intrinsic Function

Description

Test whether a string is lexically less than another string based on the ASCII collating sequence.

Class

Elemental function.

Syntax

```
result = LLT ( STRING_A , STRING_B )
```

Argument(s)

STRING_A

STRING_A shall be of type default CHARACTER.

STRING_B

STRING_B shall be of type default CHARACTER.

Result

The result is of type default LOGICAL. Its value is true if *STRING_A* precedes *STRING_B* in the ASCII collating sequence; otherwise the result is false. If both strings are of zero length the result is false.

Example

```
logical l
l = llt('elephant', 'horse') ! l is assigned the value true
```

2.309 LNBLNK Service Function

Description

Locates the position of the last nonblank character in a *string*.

Syntax

```
iy = LNBLNK ( string )
```

Argument(s)

string

Default CHARACTER scalar. String to be searched.

Result

Default INTEGER scalar. Index of the last nonblank character in *string*. Returns 0 if *string* consists of all blank characters.

Example

```
use service_routines,only:lnblnk
character(len=10) :: ch1
integer :: i
read (*,*) ch1
i = lnblnk(ch1)
write(*,fmt='(a)') ch1(1:i)
end
```

2.310 LOC Intrinsic Function

Description

Get the memory address.

Class

Utility function.

Syntax

```
result = LOC ( X )
```

Argument(s)

X

X can be of any type. It is the name for which to return an address. An array with a vector subscript cannot be specified.

Result

Eight-byte INTEGER scalar.

Example

```
i = loc(a) ! get the address of a
```

2.311 LOCK Statement

Description

The LOCK statement locks lock variable.

Syntax

```
LOCK ( lock-variable [ , lock-stat-list ] )
```

Where:

lock-variable is a lock variable . It must be of type LOCK_TYPE. See "1.18.3 LOCK_TYPE Type" for LOCK_TYPE.

lock-stat-list is a comma-separated list of

```
ACQUIRED_LOCK = scalar-logical-variable    or
sync-stat
```

where *sync-stat* is

```
STAT = stat-variable                       or
ERRMSG = errmsg-variable
```

No specifier must appear in *lock-stat-list* repeatedly. See "1.18.2 Synchronization Status Specifier" for *stat-variable* and *errmsg-variable*.

Remarks

Lock variable is locked by an image if it was locked by the image and it has not been unlocked by the image.

A lock variable is locked by the image that executed a LOCK statement without ACQUIRED_LOCK specifier. If lock variable is already locked by another image, the variable is defined by the LOCK statement when it is unlocked.

When lock variable is unlocked, it is locked by the image that executed a LOCK statement with ACQUIRED_LOCK specifier, and then a variable specified in ACQUIRED_LOCK specifier is defined with the value true. If a lock variable is already locked by another image, execution of a LOCK statement does not make the variable locked, and ACQUIRED_LOCK variable is defined with the value false.

If a lock variable is unlocked by the execution of UNLOCK statement on image M and then locked by the execution of LOCK statement on image T, segments ("1.18.1 Segment") preceding the UNLOCK statement on image M precede segments following the LOCK statement on image T. LOCK statements that do not lock variable do not affect segment ordering.

If a lock variable in a LOCK statement is already locked by executing image, an error condition occurs. If an error condition occurs during the execution of a LOCK statement, the values of the lock variable and the ACQUIRED_LOCK variable is not changed.

Example

```
use iso_fortran_env, only : lock_type
type(lock_type) :: x[*]
if (this_image() == 4) then
  lock(x)     ! lock variable "x" is locked by image 4
end if
end
```

2.312 LOG Intrinsic Function

Description

Natural logarithm.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
LOG	---	1	REAL or COMPLEX	REAL or COMPLEX
	LOG		Single-prec REAL	Single-prec REAL
	ALOG		Single-prec REAL	Single-prec REAL
	DLOG		Double-prec REAL	Double-prec REAL
	QLOG		Quad-prec REAL	Quad-prec REAL
	CLOG		Single-prec COMPLEX	Single-prec COMPLEX
	CDLOG		Double-prec COMPLEX	Double-prec COMPLEX
	CQLOG		Quad-prec COMPLEX	Quad-prec COMPLEX

result = LOG (*X*)

Argument(s)

X

X shall be of type REAL or COMPLEX. If *X* is REAL, it shall be greater than zero. If *X* is COMPLEX, it shall not be equal to zero.

Result

The result is of the same type and kind as *X*. Its value is equal to a REAL representation of $\log_e X$ if *X* is REAL. Its value is equal to the principal value with imaginary part ω in the range $-\pi < \omega <= \pi$ if *X* is COMPLEX.

Remarks

LOG, ALOG, DLOG, QLOG, CLOG, CDLOG, and CQLOG evaluate the natural logarithm of REAL or COMPLEX data.

The generic name, LOG, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
x = log (3.7)
```

2.313 LOG10 Intrinsic Function

Description

Common logarithm.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
LOG10	---	1	REAL	REAL
	LOG10		Single-prec REAL	Single-prec REAL
	ALOG10		Single-prec REAL	Single-prec REAL
	DLOG10		Double-prec REAL	Double-prec REAL
	QLOG10		Quad-prec REAL	Quad-prec REAL

result = LOG10 (*X*)

Argument(s)

X

X shall be of type REAL. The value of *X* shall be greater than zero.

Result

The result is of the same type and kind as *X*. Its value is equal to a REAL representation of $\log_{10} X$.

Remarks

LOG10, ALOG10, DLOG10, and QLOG10 evaluate the logarithm to the base 10 of REAL data.

The generic name, LOG10, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
x = log10 (3.7)
```

2.314 LOG2 Intrinsic Function

Description

Logarithm to the base 2.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
LOG2	---	1	REAL	REAL
	LOG2		Single-prec REAL	Single-prec REAL
	ALOG2		Single-prec REAL	Single-prec REAL
	DLOG2		Double-prec REAL	Double-prec REAL
	QLOG2		Quad-prec REAL	Quad-prec REAL

```
result = LOG2 ( X )
```

Argument(s)

X

X shall be of type REAL. The value of *X* shall be greater than zero.

Result

The result is of the same type and kind as *X*. Its value is equal to a REAL representation of $\log_2 X$.

Remarks

LOG2, ALOG2, DLOG2, and QLOG2 evaluate the logarithm to the base 2 of REAL data.

The generic name, LOG2, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
x = log2 (3.7)
```

2.315 LOGICAL Intrinsic Function

Description

Convert between kinds of LOGICAL.

Class

Elemental function.

Syntax

```
result = LOGICAL ( L [ , KIND ] )
```

Required Argument(s)

L

L shall be of type LOGICAL.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type LOGICAL. If *KIND* is omitted, the function result type is a default LOGICAL.

The result value is that of *L*.

Example

```
logical(kind=2) :: l
l = logical (.true.,2)  ! l is assigned the value
                        ! true with kind 2
```

2.316 LOGICAL Type Declaration Statement

Description

The LOGICAL type declaration statement declares entities of type LOGICAL. See "2.469 Type Declaration Statement" for type declaration statement.

Syntax

```
LOGICAL [ kind-selector ] [ [ , attr-spec ] ... :: ] entity-decl-list
```

2.317 LONG Service Function

Description

Converts two-byte INTEGER value to type integer.

Syntax

```
iy = LONG ( ix )
```

Argument(s)

ix

Two-byte INTEGER scalar. Value to be converted.

Result

Default INTEGER. Value of *ix* with type default INTEGER. The upper 16 bits of the result are zeros and the lower 16 are equal to *ix*.

Example

```
use service_routines,only:long
integer(2) :: ix
integer :: ix4
ix4 = long(ix)
end
```

2.318 LRSHFT Intrinsic Function

Description

Logical right shift.

Class

Elemental function.

Syntax

```
result = LRSHFT ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. Its value shall be 0 or positive and less than to the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* logical right shifted by *SHIFT* positions.

Example

```
i = lrshft(7,2)    ! i is assigned the value 1
```

2.319 LSHIFT Intrinsic Function

Description

Logical left shift.

Class

Elemental function.

Syntax

```
result = LSHIFT ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. Its value shall be 0 or positive and less than to the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* logical left shifted by *SHIFT* positions.

Example

```
i = lshift(7,2)    ! i is assigned the value 28
```

2.320 LSTAT Service Function

Description

Returns information about a file.

Syntax

```
iy = LSTAT ( name , status )
```

Argument(s)

name

Default CHARACTER scalar. A file name.

status

Default INTEGER and rank one. Its size shall be at least 13. The values returned in *status* are as follows:

```
status(1)  :  File mode.
status(2)  :  Inode number.
status(3)  :  ID of device containing a directory entry for this file.
status(4)  :  ID of device. This entry is defined only for char special or block special files.
status(5)  :  Number of links.
status(6)  :  User ID of the file's owner.
status(7)  :  Group ID of the file's group.
status(8)  :  File size in bytes.
status(9)  :  Time of last access.
status(10) :  Time of last data modification.
status(11) :  Time of last file status change.
status(12) :  Preferred I/O block size.
status(13) :  Number of 512 byte blocks allocated.
```

Result

Default INTEGER scalar. Zero if successful; otherwise, a system error code.

Example

```
use service_routines,only:lstat
integer :: st(13)
print *,lstat('f90.dat',st)
end
```

2.321 LSTAT64 Service Function

Description

Returns information about a file.

Syntax

```
iy = LSTAT64 ( name , status )
```

Argument(s)

name

Default CHARACTER scalar. A file name.

status

Eight-byte INTEGER and rank one. Its size shall be at least 13. The values returned in *status* are as follows:

```
status(1)  :  File mode.
status(2)  :  Inode number.
status(3)  :  ID of device containing a directory entry for this file.
status(4)  :  ID of device. This entry is defined only for char special or block special files.
status(5)  :  Number of links.
status(6)  :  User ID of the file's owner.
status(7)  :  Group ID of the file's group.
status(8)  :  File size in bytes.
status(9)  :  Time of last access.
status(10) :  Time of last data modification.
status(11) :  Time of last file status change.
status(12) :  Preferred I/O block size.
status(13) :  Number of 512 byte blocks allocated.
```

Result

Default INTEGER scalar. Zero if successful; otherwise, a system error code.

Example

```
use service_routines,only:lstat64
integer(kind=8) :: st(13)
print *,lstat64('f90.dat',st)
end
```

2.322 LTIME Service Subroutine

Description

Returns the local mean time in an array of time elements.

Syntax

```
CALL  LTIME ( time , t )
```

Argument(s)

time

Default INTEGER scalar. Numeric time data to be formatted. Number of seconds since 00:00:00 Greenwich mean time, January 1, 1970.

t

Default INTEGER and rank one. Its size shall be at least 9. The values returned in *t* are as follows:

Element	Value
t(1)	Seconds after the minute (0-59)
t(2)	Minutes after the hour (0-59)
t(3)	Hours since midnight (0-23)
t(4)	Day of month (1-31)
t(5)	Month since January (0-11)
t(6)	Year since 1900
t(7)	Days since Sunday (0-6)
t(8)	Days since January (0-365)
t(9)	Daylight saving flag (0 if standard time, 1 if daylight saving time)

Example

```
use service_routines,only:ltime,time
integer :: t(9)
call ltime(time(),t)
write(6,fmt="(1x,9i4)") t
end
```

2.323 MALLOC Service Function

Description

Allocates a block of memory.

Syntax

```
iy = MALLOC ( size )
```

Argument(s)

size

Eight-byte INTEGER scalar.

Result

Eight-byte INTEGER scalar. Starting address of the allocated memory.

Remarks

The allocated memory is not initialized by the Fortran system.

Example

```
use service_routines,only:free,malloc
integer(8) :: i
i = malloc(20_8)
  ...
call free(i)
end
```

2.324 MAP Statement

Description

The MAP statement begins a block of union declaration in derived type definition by STRUCTURE statement. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
MAP
```

Example

```
structure /complex_element/
  union
    map
      real :: real,imag
    end map
    map
      complex :: complex
    end map
  end union
end structure
record /complex_element/ x
x%real = 2.0
x%imag = 3.0
print *,x%complex     ! complex has the value (2.0,3.0)
```

2.325 MASKL Intrinsic Function

Description

Leftmost *I* bits set to 1 and the remaining bits set to 0.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
MASKL	---	1	INTEGER	Default INTEGER
		2	INTEGER, INTEGER	INTEGER

```
result = MASKL ( I [, KIND ]  )
```

Argument(s)

I

I shall be of type INTEGER. It shall be nonnegative and less than or equal to the number of bits of result type.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is of the INTEGER type. If *KIND* is present the result kind is *KIND*, otherwise it is the default INTEGER kind.

Remarks

MASKL evaluates value of the leftmost *I* bits set to 1 and the remaining bits set to 0.

If the kind type parameter *KIND* is specified, the result type is a *KIND* type INTEGER.If *KIND* is omitted, the result type is a default INTEGER.

Example

```
k = maskl(31)    ! k is assigned the value -2
```

2.326 MASKR Intrinsic Function

Description

Rightmost *I* bits set to 1 and the remaining bits set to 0.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
MASKR	---	1	INTEGER	Default INTEGER
		2	INTEGER, INTEGER	INTEGER

```
result = MASKR ( I [, KIND ]  )
```

Argument(s)

I

I shall be of type INTEGER. It shall be nonnegative and less than or equal to the number of bits of result type.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is of the INTEGER type. If *KIND* is present the result kind is *KIND*, otherwise it is the default INTEGER kind.

Remarks

MASKR evaluates value of the rightmost *I* bits set to 1 and the remaining bits set to 0.

If the kind type parameter *KIND* is specified, the result type is a *KIND* type INTEGER.If *KIND* is omitted, the result type is a default INTEGER.

Example

```
k = maskr(2)    ! k is assigned the value 3
```

2.327 MATMUL Intrinsic Function

Description

Matrix multiplication.

Class

Transformational function.

Syntax

```
result = MATMUL ( MATRIX_A , MATRIX_B )
```

Argument(s)

MATRIX_A

MATRIX_A shall be of type INTEGER, REAL, COMPLEX, or LOGICAL. It shall be array-valued and of rank one or two if *MATRIX_B* is of rank two, and of rank two if *MATRIX_B* is of rank one.

If *MATRIX_B* is one-dimensional array, *MATRIX_A* must be two-dimensional array.

MATRIX_B

MATRIX_B shall be of numerical type if *MATRIX_A* is of numerical type and of type LOGICAL if *MATRIX_A* is of type LOGICAL. It shall be array-valued and of rank one or two, if *MATRIX_A* is of rank two, and of rank two if *MATRIX_A* is of rank one. The size of the first dimension shall be the same as the size of the last dimension of *MATRIX_A*.

If *MATRIX_A* is one-dimensional array, *MATRIX_B* must be two-dimensional array.

The size of the first dimension of *MATRIX_B* must be the same as the size of the last dimension of *MATRIX_A*.

Result

If the arguments are of the same numeric type, the result is of that type. If their kinds are the same the result kind is that of the arguments. If their kind is different, the result kind is that of the argument with the greater kind parameter.

If the arguments are of different numeric type and one is of type COMPLEX, then the result is of type COMPLEX. If the arguments are of different numeric type, and neither is of type COMPLEX, the result is of type REAL.

If the arguments are of type LOGICAL, the result is of type LOGICAL. If their kinds are the same the result kind is that of the arguments. If their kind is different, the result kind is that of the argument with the greater kind parameter.

The shape of the result is as follows:

MATRIX_A	*MATRIX_B*	*result*
(*n* , *m*)	(*m* , *k*)	(*n* , *k*)
(*m*)	(*m* , *k*)	(*k*)
(*n* , *m*)	(*m*)	(*n*)

The value of the result is as follows:

If *MATRIX_A* has shape (*n*, *m*) and *MATRIX_B* has shape (*m*, *k*), the result has shape (*n*, *k*). 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* , :) * *MATRIX_B* (: , *j*)) if the arguments are of type LOGICAL.

If *MATRIX_A* has shape (*m*) and *MATRIX_B* has shape (*m*, *k*), the result has shape (*k*). 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* (:) * *MATRIX_B* (: , *j*)) if the arguments are of type LOGICAL.

If *MATRIX_A* has shape (*n*, *m*) and *MATRIX_B* has shape (*m*), the result has shape (*n*). 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* , :) * *MATRIX_B* (:)) if the arguments are of type LOGICAL.

Example

```
integer a(2,3), b(3), c(2)
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                     !  represents  |1 3 5|
                     !              |2 4 6|
b = (/1,2,3/)        !  represents  [1,2,3]
c = matmul(a, b)     !  c = [22,28]
```

2.328 MAX Intrinsic Function

Description

Maximum value.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
MAX	---	>=2	One-byte INTEGER	One-byte INTEGER

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	I2MAX0		Two-byte INTEGER	Two-byte INTEGER
	IMAX0		Two-byte INTEGER	Two-byte INTEGER
	MAX		Four-byte INTEGER	Four-byte INTEGER
	MAX0		Four-byte INTEGER	Four-byte INTEGER
	JMAX0		Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER	Eight-byte INTEGER
	AMAX1		Single-prec REAL	Single-prec REAL
	DMAX1		Double-prec REAL	Double-prec REAL
	QMAX1		Quad-prec REAL	Quad-prec REAL
	---		CHARACTER	CHARACTER
---	AIMAX0		Two-byte INTEGER	Single-prec REAL
	AMAX0		Four-byte INTEGER	Single-prec REAL
	AJMAX0		Four-byte INTEGER	Single-prec REAL
	IMAX1		Single-prec REAL	Two-byte INTEGER
	MAX1		Single-prec REAL	Four-byte INTEGER
	JMAX1		Single-prec REAL	Four-byte INTEGER

result = MAX (*A1* , *A2* [, *A3* , ...])

Argument(s)

A1 , *A2* [, *A3* , ...]

The arguments shall be of type INTEGER, REAL, or CHARACTER and shall all be of the same type and kind.

Result

The result is of the same type and kind as the arguments for MAX, I2MAX0, IMAX0, MAX0, JMAX0, AMAX1, DMAX1, and QMAX1, single precision REAL for AIMAX0, AMAX0, and AJMAX0, two-byte INTEGER for IMAX1, or four-byte INTEGER for MAX1 and JMAX1. Its value is the value of the largest argument.

If the argument is the CHARACTER type, the length of the result character string is the length of the longest argument.

Remarks

These functions choose the largest value of their arguments.

If the argument is the CHARACTER type, the result is the value selected by applying the intrinsic relationship operator. In other words, the character collating sequence is applied according to the argument kind type parameter. If the selected argument is shorter than the longest argument, the right side of the result is padded with blanks to match the length of the longest argument.

For two-byte INTEGER type arguments, the result type of function AIMAX0 is single precision REAL.

For four-byte INTEGER type arguments, the result type of function AMAX0 and AJMAX0 is single precision REAL.

For single precision REAL type arguments, the result type of function IMAX1 is two-byte INTEGER.

For single precision REAL type arguments, the result type of function MAX1 and JMAX1 is four-byte INTEGER.

Except for AIMAX0, AMAX0, AJMAX0, IMAX1, MAX1 and JMAX1, the generic name, MAX, may be used with any arguments and the type of the result of each function is the same as the type of the arguments.

Example

```
character(len=7) :: c
k = max(-14,3,0,-2,19,1)     ! k is assigned the value 19
c = max("abcdefg","d","xyz") ! c is assigned the value "xyz    "
```

2.329 MAXEXPONENT Intrinsic Function

Description

Maximum binary exponent of data type.

Class

Inquiry function.

Syntax

```
result = MAXEXPONENT ( X )
```

Argument(s)

X

X shall be of type REAL. It can be scalar or array-valued.

Result

The result is a scalar default INTEGER. Its value is the largest permissible binary exponent in the data type of *X*.
The result value is as follows:

Type of *X*	The result value
Single precision REAL	128
Double precision REAL	1024
Quad precision REAL	16384

Example

```
real :: r
integer :: i
i = maxexponent (r)  ! i is assigned the value 128
```

2.330 MAXLOC Intrinsic Function

Description

Location of the first element in *ARRAY* having the maximum value of the elements identified by *MASK*.

Class

Transformational function.

Syntax

```
result = MAXLOC ( ARRAY [ , MASK , KIND ] )          or
result = MAXLOC ( ARRAY , DIM [ , MASK , KIND ] )
```

Required Argument(s)

ARRAY

ARRAY shall be of type INTEGER, REAL, or CHARACTER. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar INTEGER in the range 1 <= *DIM* <= *n*, where *n* is the rank of *ARRAY*. The corresponding dummy argument shall not be an optional dummy argument.

MASK

MASK shall be of type LOGICAL and shall be conformable with *ARRAY*.

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

If *DIM* is present, when *MASK* is omitted, all *MASK* entities are assumed to be true and the following processing is performed: The result is an array of rank *n*-1 where *n* is the rank of *ARRAY*. The result values are the locations having the maximum value along dimension *DIM*.

If *DIM* is absent, the result is an array of rank one whose element values are the values of the subscripts of the first element in *ARRAY* to have the maximum value of all of the elements of *ARRAY*.

If *MASK* is present, the elements of *ARRAY* for which *MASK* is false are not considered.

If *ARRAY* is the CHARACTER type, the association result is the value selected by applying the intrinsic relationship operator. If *ARRAY* is of size 0, all entities in the results are zero. If the values of all entities in *MASK* are false, all entities in the results are zero.

Example

```
integer, dimension(1) :: i
i = maxloc ((/3,0,4,4/)) ! i is assigned the value [3]
```

2.331 MAXVAL Intrinsic Function

Description

Maximum value of elements of an array, along a given dimension, for which a mask is true.

Class

Transformational function.

Syntax

```
result = MAXVAL ( ARRAY [ , MASK ] )          or
result = MAXVAL ( ARRAY , DIM [ , MASK ] )
```

Required Argument(s)

ARRAY

ARRAY shall be of type INTEGER, REAL, or CHARACTER. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar INTEGER in the range 1 <= *DIM* <= *n*, where *n* is the rank of *ARRAY*. The corresponding dummy argument shall not be an optional dummy argument.

MASK

MASK shall be of type LOGICAL and shall be conformable with *ARRAY*.

Result

The result is of the same type and kind as *ARRAY*. It is scalar if *DIM* is absent or if *ARRAY* has rank one; otherwise the result is an array of rank *n*-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of *ARRAY*. If *DIM* is absent, the value of the result is the maximum value of all the elements of *ARRAY*. When the *ARRAY* has a size of zero and is of type CHARACTER, the result is the value of the character string of the same length LEN(*ARRAY*) as when each character is CHAR(0,KIND=KIND(*ARRAY*)). If *DIM* is present, the value of the result is the maximum value of all elements of *ARRAY* along dimension *DIM*. If *MASK* is present, the elements of *ARRAY* for which *MASK* is false are not considered.

If *ARRAY* is the CHARACTER type, the association result is the value selected by applying the intrinsic relationship operator. In other words, the character collating sequence is applied by the argument kind type parameter.

Example

```
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
integer j(2)
! m is the array |1 3|
!               |2 4|
i = maxval(m)          ! i is assigned the value 4
j = maxval(m,dim=1)    ! j is assigned the value [2,4]
k = maxval(m,mask=m<3) ! k is assigned the value 2
```

2.332 MERGE Intrinsic Function

Description

Choose alternative values based on the value of a mask.

Class

Elemental function.

Syntax

```
result = MERGE ( TSOURCE , FSOURCE , MASK )
```

Argument(s)

TSOURCE

TSOURCE can be of any type.

FSOURCE

FSOURCE shall be of the same type and type parameters as *TSOURCE*.

MASK

MASK shall be of type LOGICAL.

Result

The result is of the same type and type parameters as *TSOURCE*. Its value is *TSOURCE* if *MASK* is true, and *FSOURCE* otherwise.

Example

```
integer, dimension (2,2) :: r
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
! m is the array |1 3|
!               |2 4|
integer, dimension (2,2) :: n = reshape((/3,3,3,3/),(/2,2/))
! n is the array |3 3|
!               |3 3|
r = merge(m,n,m<n) ! r is assigned the value |1 3|
                   !                        |2 3|
```

2.333 MERGE_BITS Intrinsic Function

Description

Merge of bits under mask.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
MERGE_BITS	---	3	INTEGER, or binary, octal or hexadecimal constant,	INTEGER

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
			INTEGER, or binary, octal or hexadecimal constant, INTEGER, or binary, octal or hexadecimal constant	

```
result = MERGE_BITS ( I , J , MASK )
```

Argument(s)

I

I shall be of type INTEGER, or a binary, octal or hexadecimal constant.

J

J shall be of type INTEGER, or a binary, octal or hexadecimal constant.

MASK

MASK shall be of type INTEGER, or a binary, octal or hexadecimal constant.

Result

The type of the result is the same as the type of the argument *I* or *J*.

Remarks

MERGE_BITS selects either other bit according to the bit value of *MASK*.

The result is same IOR (IAND (*I*,*MASK*) , IAND (*J*,NOT(*MASK*)).

If *I* and *J* are of type INTEGER, *I* and *J* shall be same kind.

If a binary, octal or hexadecimal constant is specified for either, the constant is converted to the specified type INTEGER. The binary, octal or hexadecimal constants for both shall not specify in *I* and *J*.

If *MASK* is of type INTEGER, the type shall be same kind with *I* or *J*. If *MASK* is a binary, octal or hexadecimal constant, the constant is converted to the specified type INTEGER.

The type of the result is the same as the type of the argument *I* or *J*.

Example

```
k = merge_bits(18,13,22)    ! k is assigned the value 27
```

2.334 MIN Intrinsic Function

Description

Minimum value.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
MIN	---	>=2	One-byte INTEGER	One-byte INTEGER
	I2MIN0		Two-byte INTEGER	Two-byte INTEGER
	IMIN0		Two-byte INTEGER	Two-byte INTEGER
	MIN		Four-byte INTEGER	Four-byte INTEGER
	MIN0		Four-byte INTEGER	Four-byte INTEGER
	JMIN0		Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER	Eight-byte INTEGER

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	AMIN1		Single-prec REAL	Single-prec REAL
	DMIN1		Double-prec REAL	Double-prec REAL
	QMIN1		Quad-prec REAL	Quad-prec REAL
	---		CHARACTER	CHARACTER
---	AIMIN0		Two-byte INTEGER	Single-prec REAL
	AMIN0		Four-byte INTEGER	Single-prec REAL
	AJMIN0		Four-byte INTEGER	Single-prec REAL
	IMIN1		Single-prec REAL	Two-byte INTEGER
	MIN1		Single-prec REAL	Four-byte INTEGER
	JMIN1		Single-prec REAL	Four-byte INTEGER

```
result = MIN ( A1 , A2 [ , A3 , ... ] )
```

Argument(s)

A1 , *A2* [, *A3* , ...]

The arguments shall be of type INTEGER, REAL, or CHARACTER and shall all be of the same type and kind.

Result

The result is of the same type and kind as the arguments for MIN, I2MIN0, IMIN0, MIN0, JMIN0, AMIN1, DMIN1, and QMIN1, single precision REAL for AIMIN0, AMIN0, and AJMIN0, two-byte INTEGER for IMIN1, or four-byte INTEGER for MIN1 and JMIN1. Its value is the value of the smallest argument.

If the argument is the CHARACTER type, the length of the result character string is the length of the longest argument.

Remarks

These functions choose the smallest value of their arguments.

If the argument is the CHARACTER type, the result is the value selected by applying the intrinsic relationship operator. In other words, the character collating sequence is applied according to the argument kind type parameter. If the selected argument is shorter than the longest argument, the right side of the result is padded with blanks to match the length of the longest argument.

For two-byte INTEGER type arguments, the result type of function AIMIN0 is single precision REAL.

For four-byte INTEGER type arguments, the result type of function AMIN0 and AJMIN0 is single precision REAL.

For single precision REAL type arguments, the result type of function IMIN1 is two-byte INTEGER.

For single precision REAL type arguments, the result type of function MIN1 and JMIN1 is four-byte INTEGER.

Except for AIMIN0, AMIN0, AJMIN0, IMIN1, MIN1, and JMIN1, the generic name, MIN, may be used with any arguments and the type of the result of each function is the same as the type of the arguments.

Example

```
character(len=7) :: c
k = min(-14,3,0,-2,19,1)       ! k is assigned the value -14
c = min("abc","defghij","xyz") ! c is assigned "abc    "
```

2.335 MINEXPONENT Intrinsic Function

Description

Minimum binary exponent of data type.

Class

Elemental function.

Syntax

```
result = MINEXPONENT ( X )
```

Argument(s)

X

X shall be of type REAL. It can be scalar or array-valued.

Result

The result is a scalar default INTEGER. Its value is the most negative permissible binary exponent in the data type of *X*.
The result value is as follows:

Type of *X*	The result value
Single precision REAL	-125
Double precision REAL	-1021
Quad precision REAL	-16381

Example

```
real :: r
integer :: i
i = minexponent (r)  ! i is assigned the value -125
```

2.336 MINLOC Intrinsic Function

Description

Location of the first element in *ARRAY* having the minimum value of the elements identified by *MASK*.

Class

Transformational function.

Syntax

```
result = MINLOC ( ARRAY [ , MASK , KIND ] )          or
result = MINLOC ( ARRAY , DIM [ , MASK , KIND ] )
```

Required Argument(s)

ARRAY

ARRAY shall be of type INTEGER, REAL, or CHARACTER. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar INTEGER in the range 1 <= *DIM* <= *n*, where *n* is the rank of *ARRAY*. The corresponding dummy argument shall not be an optional dummy argument.

MASK

MASK shall be of type LOGICAL and shall be conformable with *ARRAY*.

KIND

KIND shall be of type scalar INTEGER initialization expression.

Result

If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

If *DIM* is present, when *MASK* is omitted, all *MASK* entities are assumed to be true and the following processing is performed: The result is an array of rank *n*-1 where *n* is the rank of *ARRAY*. The result values are the locations having the minimum value along dimension *DIM*.

If *DIM* is absent, the result is an array of rank one whose element values are the values of the subscripts of the first element in *ARRAY* to have the minimum value of all of the elements of *ARRAY*.

If *MASK* is present, the elements of *ARRAY* for which *MASK* is false are not considered.

If *ARRAY* is the CHARACTER type, the association result is the value selected by applying the intrinsic relationship operator. If *ARRAY* is of size 0, all entities in the results are zero. If the values of all entities in *MASK* are false, all entities in the results are zero.

Example

```
integer, dimension(1) :: i
i = minloc ((/3,0,4,4/)) ! i is assigned the value [2]
```

2.337 MINVAL Intrinsic Function

Description

Minimum value of elements of an array, along a given dimension, for which a mask is true.

Class

Transformational function.

Syntax

```
result = MINVAL ( ARRAY [ , MASK ] )          or
result = MINVAL ( ARRAY , DIM [ , MASK ] )
```

Required Argument(s)

ARRAY

ARRAY shall be of type INTEGER, REAL, or CHARACTER. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar INTEGER in the range 1 <= *DIM* <= *n*, where *n* is the rank of *ARRAY*. The corresponding dummy argument shall not be an optional dummy argument.

MASK

MASK shall be of type LOGICAL and shall be conformable with *ARRAY*.

Result

The result is of the same type and kind as *ARRAY*. It is scalar if *DIM* is absent or if *ARRAY* has rank one; otherwise the result is an array of rank *n*-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of *ARRAY*. When the *ARRAY* has a size of zero and is of type CHARACTER, the result is the value of the character string of the same length LEN(*ARRAY*) as when each character is CHAR(255,KIND=1). If *DIM* is absent, the value of the result is the minimum value of all the elements of *ARRAY*. If *DIM* is present, the value of the result is the minimum value of all elements of *ARRAY* along dimension *DIM*. If *MASK* is present, the elements of *ARRAY* for which *MASK* is false are not considered.

If *ARRAY* is the CHARACTER type, the association result is the value selected by applying the intrinsic relationship operator. In other words, the character collating sequence is applied by the argument kind type parameter.

Example

```
integer j(2)
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
! m is the array |1 3|
!                |2 4|
i = minval(m)          ! i is assigned the value 1
j = minval(m,dim=1)    ! j is assigned the value [1,3]
k = minval(m,mask=m<3) ! k is assigned the value 1
```

2.338 MOD Intrinsic Function

Description

Remainder.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
MOD	---	2	One-byte INTEGER, One-byte INTEGER	One-byte INTEGER
	I2MOD		Two-byte INTEGER, Two-byte INTEGER	Two-byte INTEGER
	IMOD		Two-byte INTEGER, Two-byte INTEGER	Two-byte INTEGER
	MOD		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	JMOD		Four-byte INTEGER, Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER, Eight-byte INTEGER	Eight-byte INTEGER
	AMOD		Single-prec REAL, Single-prec REAL	Single-prec REAL
	DMOD		Double-prec REAL, Double-prec REAL	Double-prec REAL
	QMOD		Quad-prec REAL, Quad-prec REAL	Quad-prec REAL

```
result = MOD ( A , P )
```

Argument(s)

A

A shall be of type INTEGER or REAL.

P

P shall be of the same type and kind as *A*. Its value shall not be zero.

Result

The result is the same type and kind as *A*. Its value is *A* - INT(*A* / *P*)**P*.

Remarks

The generic name, MOD, may be used with any argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = mod(23.0,4.0) ! r is assigned the value 3.0
i = mod(-23,4)    ! i is assigned the value -3
j = mod(23,-4)    ! j is assigned the value 3
k = mod(-23,-4)   ! k is assigned the value -3
```

2.339 MODULE Statement

Description

The MODULE statement begins a module. See "1.11.2 Modules" for module.

Syntax

```
MODULE module-name
```

Where:

module-name is the name of the module.

Remarks

module-name is global entities of a program, it shall not be used to identify other program units, external procedures, and common blocks in the same program. Also it shall not be used to identify a local entity in the module.

Example

```
module mod
  implicit none
  type mytype   ! mytype available anywhere mod is used
    real :: a, b(2,4)
    integer :: n,o,p
  end type mytype
end module mod
subroutine zee()
  use mod
  implicit none
  type (mytype) bee, dee
  ...
end subroutine zee
```

2.340 MODULO Intrinsic Function

Description

Modulo.

Class

Elemental function.

Syntax

```
result = MODULO ( A , P )
```

Argument(s)

A

A shall be of type INTEGER or REAL.

P

P shall be of the same type and kind as *A*. Its value shall not be zero.

Result

The result is the same type and kind as *A*. If *A* is a REAL, the result value is *A*-FLOOR(*A* / *P*)**P*.
If *A* is an INTEGER, MODULO(*A* , *P*) has the value *r* such that *A*=*q* * *P* + *r* , where *q* is an INTEGER and *r* is nearer to zero than *P*.

Example

```
r = modulo(23.0,4.0) ! r is assigned the value 3.0
i = modulo(-23,4)    ! i is assigned the value 1
```

```
j = modulo(23,-4)    ! j is assigned the value -1
k = modulo(-23,-4)   ! k is assigned the value -3
```

2.341 MOVE_ALLOC Intrinsic Subroutine

Description

Moves an allocation from one allocatable object to another.

Class

Pure subroutine.

Syntax

```
CALL MOVE_ALLOC ( FROM , TO )
```

Argument(s)

FROM

FROM may be of any type and rank. It shall be allocatable. It is an INTENT(INOUT) argument.

TO

TO shall be type compatible with *FROM* and have the same rank and corank. It shall be allocatable. It is an INTENT(OUT) argument. If *TO* is character type, it's character length parameter shall be same with *FROM*.

Remarks

The allocation status of *TO* becomes unallocated if *FROM* is deallocated on entry to MOVE_ALLOC. Otherwise, *TO* becomes allocated with array bounds, cobounds, and value identical to those that *FROM* had on entry to MOVE_ALLOC.

If *TO* has the TARGET attribute, any pointer associated with *FROM* on entry to MOVE_ALLOC becomes correspondingly associated with *TO*. If *TO* does not have the TARGET attribute, the pointer association status of any pointer associated with *FROM* on entry becomes undefined.

The allocation status of *FROM* becomes unallocated.

When a MOVE_ALLOC is executed for which an argument is a coarray, there is an implicit synchronization of all images. On each image, execution of the segment following the statement is delayed until all other images have executed the same statement.

Example

```
integer, allocatable :: a(:), b(:)
allocate(a(1:10))
allocate(b(1:100))
call move_alloc(b, a)  ! "a" is deallocated, and becomes allocated with
                       ! array bounds of "b"
                       ! "b" becomes deallocated
```

2.342 MTOIE Service Subroutine

Description

IBM370-format floating-point data is converted to IEEE-format floating-point data. For information on IEEE-format floating-point data and information on IBM370-format floating-point data, see "Fortran User's Guide".

Syntax

```
CALL  MTOIE ( r1 , r2 , type , retcd )
```

Argument(s)

r1

Single precision REAL or double precision REAL scalar. IBM370-format floating-point data to convert.

r2

Single precision REAL or double precision REAL scalar. Returns IEEE-format floating-point data.

type

Default INTEGER scalar. The converted type is as follows:

```
=0  :  Single precision REAL
=1  :  Double precision REAL
```

retcd

Default INTEGER scalar. Return code as follows:

```
=0  :  Processing ended normally.
=4  :  1 to 3 bits in the mantissa of floating-point data were lost during conversion.
=8  :  Floating-point overflow or underflow, Inf or NaN was detected.
=12 :  The third argument type is invalid.
```

Example

```
use service_routines,only:mtoie
real(kind=8) :: mdata
real(kind=8) :: ie3data
integer :: ret1,cltype=1
data mdata/z'8000000000000000'/
data ie3data/z'7ff0000000000000'/
call mtoie(mdata,ie3data,cltype,ret1)
end
```

2.343 MVBITS Intrinsic Subroutine

Description

Copy a sequence of bits from one INTEGER data object to another.

Class

Elemental subroutine.

Syntax

```
CALL  MVBITS ( FROM , FROMPOS , LEN , TO , TOPOS )
```

Argument(s)

FROM

FROM shall be of type INTEGER. It is an INTENT(IN) argument.

FROMPOS

FROMPOS shall be of type INTEGER and shall be non-negative. It is an INTENT(IN) argument. *FROMPOS* + *LEN* shall be less than or equal to BIT_SIZE(*FROM*).

FROMPOS shall be greater than or equal zero.

LEN

LEN shall be of type INTEGER and shall be non-negative. It is an INTENT(IN) argument.

TO

TO shall be a variable of type INTEGER with the same kind as *FROM*. It can be the same variable as *FROM*. It is an INTENT(INOUT) argument. *TO* is set by copying *LEN* bits, starting at position *FROMPOS*, from *FROM*, to *TO*, starting at position *TOPOS*.

TOPOS

TOPOS shall be of type INTEGER and shall be non-negative. It is an INTENT(IN) argument. *TOPOS* + *LEN* shall be less than or equal to BIT_SIZE(*TO*).

TOPOS shall be greater than or equal zero.

Example

```
i = 17
j = 3
call mvbits (i,3,3,j,1) ! j is assigned the value 5
```

2.344 NAMELIST Statement

Description

The NAMELIST statement specifies a list of variables which can be referred to by one name for the purpose of performing input/output.

Syntax

```
NAMELIST / namelist-group-name / namelist-group-object-list    &
&   [ [ , ] / namelist-group-name / namelist-group-object-list ] ...
```

Where:

namelist-group-name is the name of a namelist group.

The *namelist-group-name* shall not be a name made accessible by use association.

Any *namelist-group-name* may occur in more than one NAMELIST statement in a scoping unit. The *namelist-group-object-list* following each successive appearance of the same *namelist-group-name* in a scoping unit is treated as a continuation of the list for that *namelist-group-name*.

namelist-group-object-list is a comma-separated list of

```
variable
```

variable is the name of a variable.

Remarks

A *variable* shall not be the name of a variable of a type that has an ultimate component that is a pointer, an object of a derived type containing an ultimate component that is an allocatable, or an object of a derived type containing an ultimate component that has length type parameter.

A namelist group object shall either be accessed by use or host association or shall have its type, type parameters, and shape specified by previous specification statements in the same scoping unit or by the implicit typing rules in effect for the scoping unit. If a namelist group object is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm the implied type and type parameters.

A namelist group object may be a member of more than one namelist group.

If a *namelist-group-name* has the PUBLIC attribute, no item in the *namelist-group-object-list* shall have the PRIVATE attribute or have private components.

The order in which the variables appear in a NAMELIST statement determines the order in which the variable's values will appear on output.

Example

```
namelist /mylist/ x, y, z
```

2.345 NARGS Service Function

Description

Returns the number of command-line arguments, including the command.

Syntax

```
iy = NARGS ( )
```

Result

Default INTEGER scalar. The number of command-line arguments, including the command.

Example

```
use service_routines,only:nargs
print *,nargs()
end
```

2.346 NEAREST Intrinsic Function

Description

Nearest number of a given data type in a given direction.

Class

Elemental function.

Syntax

```
result = NEAREST ( X , S )
```

Argument(s)

X

X shall be of type REAL.

S

S shall be of type REAL and shall be non-zero.

Result

The result is of the same type and kind as *X*. Its value is the nearest distinct number, in the data type of *X*, from *X* in the direction of the infinity with the same sign as *S*.

Example

```
a = nearest (34.3, -2.0)
```

2.347 NEW_LINE Intrinsic Function

Description

Returns a newline character.

Class

Inquiry function.

Syntax

```
result = NEW_LINE ( A )
```

Argument(s)

A

A shall be of type CHARACTER. It can be scalar or array-valued.

Result

Result is CHAR (10, KIND (*A*)).

The result is of type CHARACTER scalar. The kind type parameter of the result is the same as *A*, and the character length is 1.

Example

```
character(len=50) :: char_string
character(len=1) :: nl
nl = new_line(char_string)  ! nl is assigned the value of CHAR(10)
```

2.348 NINT Intrinsic Function

Description

Nearest INTEGER.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
NINT	---	1 or 2	REAL [,INTEGER]	INTEGER
	NINT	1	Single-prec REAL	Default INTEGER
	JNINT		Single-prec REAL	Default INTEGER
	IDNINT		Double-prec REAL	Default INTEGER
	JIDNNT		Double-prec REAL	Default INTEGER
	IQNINT		Quad-prec REAL	Default INTEGER
I2NINT	---	1	REAL	Two-byte INTEGER
	ININT		Single-prec REAL	Two-byte INTEGER
	IIDNNT		Double-prec REAL	Two-byte INTEGER

```
result = NINT ( A [ , KIND ] )
```

Required Argument(s)

A

A shall be of type REAL.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is of type INTEGER. If *KIND* is present the result kind is *KIND*; otherwise it is the default INTEGER kind. If *A* > 0, the result has the value INT(*A* + 0.5); if *A* <= 0, the result has the value INT(*A* - 0.5).

Remarks

NINT, I2NINT, JNINT, IDNINT, JIDNNT, IQNINT, ININT, and IIDNNT evaluate the nearest number. If *A* > 0 , the result has the value INT(*A* + 0.5). If *A* < 0 , the result has the value INT(*A* - 0.5).

The generic name, NINT, may be used with any REAL argument. The kind type parameter of the result of each function is *KIND* or default INTEGER except for I2NINT, JNINT and IIDNNT. If the kind type parameter *KIND* is specified, the function result type is a *KIND* type INTEGER. If *KIND* is omitted, the function result type is a default INTEGER.

The generic name, I2NINT, may be used with any real argument. The kind type parameter of the result of each function is two-byte INTEGER.

Example

```
i = nint (7.73)  ! i is assigned the value 8
i = nint (-4.2)  ! i is assigned the value -4
i = nint (-7.5)  ! i is assigned the value -8
i = nint (2.50)  ! i is assigned the value 3
```

2.349 NOT Intrinsic Function

Description

Bit-wise logical complement.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
NOT	---	1	One-byte INTEGER	One-byte INTEGER
	INOT		Two-byte INTEGER	Two-byte INTEGER
	NOT		Four-byte INTEGER	Four-byte INTEGER
	JNOT		Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER	Eight-byte INTEGER

```
result = NOT ( I )
```

Argument(s)

I

I shall be of type INTEGER.

Result

The result is of the same type and kind as *I*. Its value is the value of *I* with each of its bits complemented (zeros changed to ones and ones changed to zeros).

Example

```
i = not(5) ! i is assigned the value -6
```

2.350 NULL Intrinsic Function

Description

Returns a disassociated pointer or an unallocated allocatable object.

Class

Transformational function.

Syntax

```
result = NULL ( [ MOLD ] )
```

Optional Argument(s)

MOLD

Must be a pointer or an allocatable. Any type is acceptable.

If *MOLD* is a pointer, the pointer associate status can be undefined, disassociated, or associated.

If *MOLD* is an allocatable, the allocate status can be allocated or unallocated.

Result

A disassociated pointer or unallocated allocatable of the same type, type parameters, and rank as *MOLD* if present, otherwise, follows:

Appearance of NULL( )	Type, type parameters, and rank of result
right side of a pointer assignment statement	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

Example

```
real, pointer, dimension(:) :: a => null() ! a is disassociated
```

2.351 NULLIFY Statement

Description

The NULLIFY statement disassociates pointers.

Syntax

```
NULLIFY ( pointer-object-list )
```

Where:

pointer-object-list is a comma-separated list of

```
variable-name              or
structure-component        or
proc-pointer-name
```

variable-name is the name of variable that has the POINTER attribute.

proc-pointer-name is the name of procedure pointer.

structure-component is the structure component that has the POINTER attribute.

Example

```
real, pointer :: a, b, c
real, target :: t, u, v
a=>t; b=>u; c=>v   ! a, b, and c are associated
nullify (a, b, c)  ! a, b, and c are disassociated
```

2.352 NUM_IMAGES Intrinsic Function

Description

Returns the number of images.

Class

Transformational function.

Syntax

```
result = NUM_IMAGES ( )
```

Result

A default integer scalar. Its value is the number of images.

Example

```
k = num_images() ! the number of images is assigned to variable k
```

2.353 OMP_LIB Nonstandard Intrinsic Module

Description

This nonstandard intrinsic module enables the reference to the named constant and explicit interface of OpenMP standard.

2.354 OPEN Statement

Description

The OPEN statement connects or reconnects an external file and an input/output unit.

Syntax

```
OPEN ( connect-spec-list )
```

Where:

connect-spec-list is a comma-separated list of

```
[ UNIT = ] external-file-unit        or
IOSTAT = io-stat                     or
ERR = err-label                      or
FILE = file-name-expr                or
STATUS = status                      or
ACCESS = access                      or
FORM = form                          or
RECL = recl                          or
BLANK = blank                        or
POSITION = position                  or
ACTION = action                      or
DELIM = delim                        or
PAD = pad                            or
TOTALREC = totalrec                  or
BLOCKSIZE = blocksize                or
CARRIAGECONTROL = carriagecontrol    or
CONVERT = convert                    or
ASYNCHRONOUS = asynchronous          or
DECIMAL = decimal                    or
ENCODING = encoding                  or
IOMSG = iomsg                        or
ROUND = round                        or
SIGN = sign
```

An *external-file-unit* shall be specified. If the optional characters UNIT= are omitted from the *external-file-unit* specifier, the *external-file-unit* specifier shall be the first item in the *connect-spec-list*.

external-file-unit is a scalar INTEGER expression that evaluates to the input/output unit number of an external file.

io-stat is a scalar INTEGER variable that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, and zero otherwise.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the OPEN statement. If an error condition occurs during execution of the OPEN statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

file-name-expr is a scalar CHARACTER expression that evaluates to the name of a file. Any leading and trailing blanks are ignored. If this specifier is omitted and the unit is not connected to a file, the STATUS= specifier shall be specified with a value of SCRATCH; in this case, the connection is made to a nonnamed file or the named file that is specified by environment variable for execution (see "Fortran User's Guide").

status is a scalar default CHARACTER expression. It shall evaluate to NEW if the file does not exist and is to be created; REPLACE if the file is to overwrite an existing file of the same name or create a new one if the file does not exist; SCRATCH if the file is to be deleted at the end of the program or the execution of a CLOSE statement; OLD or SHR, if the file is to be opened but not replaced; and UNKNOWN otherwise. If *status* evaluates to NEW, REPLACE, or SHR, the FILE= specifier shall be present. If *status* evaluates to SCRATCH, the FILE= specifier shall be absent. If *status* evaluates to OLD, the FILE= specifier shall be present unless the unit is currently connected and the file connected to the unit exists. If UNKNOWN is specified, it is the same as NEW where FILE= specifier is present and the file is not exist, the same as OLD where FILE= specifier is present and the file is exist or FILE= specifier is not present and environment variable for execution is effective, and the same as SCRATCH where FILE= specifier is not present and environment variable for execution is ineffective. The default is UNKNOWN.

access is a scalar default CHARACTER expression. It shall evaluate to SEQUENTIAL if the file is to be connected for sequential access, DIRECT if the file is to be connected for direct access, STREAM if the file is to be connected for stream access, and APPEND if the file is to be connected for sequential access and it is to be positioned before the endfile record if one exists and at the file terminal point otherwise; the same as POSITION='APPEND'. The default value is SEQUENTIAL.

form is a scalar default CHARACTER expression. It shall evaluate to FORMATTED if the file is to be connected for formatted input/output, UNFORMATTED if the file is to be connected for unformatted input/output, and BINARY if the file is to be connected for binary input/output. The default value is UNFORMATTED, for a file connected for direct access, and FORMATTED, for a file connected for sequential access.

recl is a scalar default INTEGER expression. It shall evaluate to the record length in bytes for a file connected for direct access, or the maximum record length in bytes for a file connected for sequential access. *recl* cannot specify if the file connected for stream access. The RECL= specifier shall be present if the file connected for direct access. If the file connected for sequential access and the RECL= specifier is absent, the default value is 2147483647.

blank is a scalar default CHARACTER expression. It shall evaluate to NULL if null blank control is to be used and ZERO if zero blank control is to be used. The default value is NULL. The BLANK= specifier is only permitted for a file being connected for formatted input/output.

position is a scalar default CHARACTER expression. It shall evaluate to REWIND if the newly opened sequential access file is to be positioned at its initial point; APPEND if it is to be positioned before the endfile record if one exists and at the file terminal point otherwise; and ASIS if the position is to be left unchanged. The default is ASIS. The POSITION= specifier is only permitted for a file being connected for sequential access or stream access.

action is a scalar default CHARACTER expression. It shall evaluate to READ if the file is to be connected for input only, WRITE if the file is to be connected for output only, and READWRITE or BOTH if the file is to be connected for input and output. The default value is READWRITE.

delim is a scalar default CHARACTER expression. It shall evaluate to APOSTROPHE if the apostrophe will be used to delimit character constants written with list-directed or namelist formatting and all internal apostrophes will be doubled, QUOTE if the quotation mark will be used and all internal quotation marks will be doubled, and NONE if neither quotation marks nor apostrophes will be used. The default value is NONE. The DELIM= specifier is permitted only for formatted files and is ignored on input.

pad is a scalar default CHARACTER expression. It shall evaluate to YES if the formatted input record is to be padded with blanks and NO otherwise. If NO is specified, the input list and the format specification shall not require more characters from a record than the record contains. The default value is YES.

totalrec is a scalar default INTEGER expression. It shall evaluate to the number of records that can be read or written in a file connected for direct access. If the file is exist, the TOTALREC= specifier is ignored. The default is 2147483647.

blocksize is a scalar default INTEGER expression. It shall evaluate to the size, in bytes, of the input/output buffer. See "Fortran User's Guide" for default value.

carriagecontrol is a scalar default CHARACTER expression. It shall evaluate to FORTRAN if the first character of a formatted sequential record is to be used for carriage control listed below, and LIST otherwise. The default value is LIST.

Character	Effective
blank	Advance to the next line and print the line
+	Do not advance to the next line before printing the line (over-print the current line)
0	Advance two lines and print the line
1	Advance to the first line of the next page and print the line
other than above	The same as blank

convert is a scalar default CHARACTER expression. It shall evaluate to LITTLE_ENDIAN if the file connected for has little endian data, BIG_ENDIAN if the file connected for has big endian data, IBM if the file connected for has IBM-370 format, and NATIVE if the file is connected for has native data of the processor.

asynchronous is a scalar default CHARACTER expression.

The ASYNCHRONOUS= specifier specifies whether an input/output statement is synchronous or asynchronous. The asynchronous value must be either 'YES' or 'NO'. A 'YES' specifies that input/output to that device can be asynchronous. A 'NO' specifies that input/output to that device cannot be asynchronous. If this specifier is omitted, 'NO' is the default.

decimal must be a scalar default CHARACTER expression. The *decimal* value must be either 'COMMA' or 'POINT'. The DECIMAL= specifier can be specified only for files connected as formatted input/output. It specifies the value of the current decimal editing mode at this connection. This mode can be changed. If this specifier is omitted from the OPEN statement that starts a connection, 'POINT' is the default.

encoding must be a scalar default CHARACTER expression. The *encoding* value must be either 'UTF-8' or 'DEFAULT'. The ENCODING= specifier can be specified only for files connected as formatted input/output. If 'UTF-8' is specified, the file encoding method is the UTF-8 prescribed by JIS X 0221-1 : 2001. This type of file is called a Unicode file, and all characters in that file are of the ISO 10646 CHARACTER type. If this specifier is omitted, 'DEFAULT' is the default.

iomsg must be a scalar default CHARACTER variable. If the error condition occurs during input/output statement execution, an explanatory message is assigned to *iomsg*.

If an error condition does not occur, the i*omsg* value does not change.

round is a scalar default CHARACTER expression, and the value must be either 'UP', 'DOWN', 'ZERO', 'NEAREST', 'COMPATIBLE', or 'PROCESSOR_DEFINED'.

The ROUND= specifier can be specified only for files connected as formatted input/output. ROUND= specifies the value of the current input/output rounding mode at this connection. This mode can be changed.

sign must be a scalar default CHARACTER expression. The value of *sign* is either 'PLUS', 'SUPPRESS', or 'PROCESSOR_DEFINED'.

The SIGN= specifier can be specified only for files connected as formatted input/output. It specifies the value of the current sign mode at this connection. This mode can be changed. If ROUND is omitted from the OPEN statement that starts the connection, 'PROCESSOR_DEFINED' is the default.

Remarks

The OPEN statement can be used to connect an existing file to an input/output unit, create a file that is preconnected, create a file and connect it to an input/output unit, or change certain characteristics of a connection between a file and an input/output unit.

If the file to be connected to the input/output unit is the same as the file to which the unit is already connected, only the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers can have values different from those currently in effect.

If a file is already connected to an input/output unit, execution of an OPEN statement on that file and a different unit is not permitted.

Example

```
open (10, file='info.dat',status='new')
```

2.355 OPTIONAL Statement

Description

The OPTIONAL statement specifies that any of the dummy arguments specified need not be associated with an actual argument when the procedure is invoked.

Syntax

```
OPTIONAL [ :: ] dummy-arg-name-list
```

Where:

dummy-arg-name-list is a comma-separated list of dummy argument names.

Remarks

An OPTIONAL statement shall appear only in the specification part of a subprogram or an interface body.

Example

```
subroutine sub(a,b)
  optional :: b
  ! b need not be provided when calling sub
   ...
```

2.356 OVERFL Service Subroutine

Description

Tests for an exponent overflow or underflow exception.

Syntax

```
CALL OVERFL ( i )
```

Argument(s)

i

Default INTEGER scalar. The OVERFL service subroutine tests for an exponent overflow or underflow exception, and returns a value that indicates the existing condition. After testing, the overflow indicator is turned off. The value of *i* is returned by the service subroutine to indicate the following:

```
=1  :   Floating-point overflow.
=2  :   No overflow or underflow.
=3  :   Floating-point underflow.
```

Example

```
use service_routines,only:overfl
integer :: i0, i1, i2
real :: huge_real = huge(f)
real :: tiny_real = tiny(f)
call overfl(i0)  ! i0 .eq. 2
call sub_overflow(huge_real)
call overfl(i1)  ! i1 .eq. 1
call overfl(i1)  ! i1 .eq. 2
call sub_underflow(tiny_real)
call overfl(i2)  ! i2 .eq. 3
end
subroutine sub_overflow(huge_real)
real, intent(inout) :: huge_real
huge_real = huge_real * 2.0       ! overflow
end
subroutine sub_underflow(tiny_real)
real, intent(inout) :: tiny_real
tiny_real = tiny_real / 1.0e+08   ! underflow
end
```

2.357 PACK Intrinsic Function

Description

Pack an array into a vector under control of a mask.

Class

Transformational function.

Syntax

```
result = PACK ( ARRAY , MASK [ , VECTOR ] )
```

Required Argument(s)

ARRAY

ARRAY can be of any type. It shall not be scalar.

MASK

MASK shall be of type LOGICAL and shall be conformable with *ARRAY*.

Optional Argument(s)

VECTOR

VECTOR shall be of the same type and kind as *ARRAY* and shall have rank one. It shall have at least as many elements as there are true elements in *ARRAY*. If *MASK* is scalar with value true, *VECTOR* shall have at least as many elements as *ARRAY*.

Result

The result is an array of rank one with the same type and kind as *ARRAY*. If *VECTOR* is present, the result size is the size of *VECTOR*. If *VECTOR* is absent, the result size is the number of true elements in *MASK* unless *MASK* is scalar with the value true, in which case the size is the size of *ARRAY*.

The value of element *i* of the result is the ith true element of *MASK*, in array-element order. If *VECTOR* is present and is larger than the number of true elements in *MASK*, the elements of the result beyond the number of true elements in *MASK* are filled with values from the corresponding elements of *VECTOR*.

However, when *MASK* is scalar and has a value of true, the resulting size is the *ARRAY* size.

Example

```
integer, dimension(3,3) :: c
integer, dimension(6) :: d
integer, dimension(9) :: e
c = reshape((/0,3,2,4,3,2,5,1,2/),(/3,3/))
! represents the array |0 4 5|
!                      |3 3 1|
!                      |2 2 2|
d = pack(c,mask=c.ne.2)! d is assigned [0 3 4 3 5 1]
e = pack(c,mask=.true.)! e is assigned [0 3 2 4 3 2 5 1 2]
```

2.358 PARAMETER Statement

Description

The PARAMETER statement specifies named constants.

Syntax

```
PARAMETER ( named-constant-def-list )
```

Where:

named-constant-def-list is a comma-separated list of

```
constant-name = init-expr
```

constant-name is the name of a constant being specified.

init-expr is an initialization expression.

Remarks

The named constant shall have its type, type parameters, and shape specified in a prior specification of the specification part or declared implicitly. If the named constant is typed by the implicit typing rules, its appearance in any subsequent specification of the specification part shall confirm this implied type and the values of any implied type parameters.

Each named constant becomes defined with the value of *init-expr* in accordance with the rules of intrinsic assignment (see "2.27 Assignment Statement").

Example

```
real :: freezing_point, conv_factor
parameter (freezing_point = 32.0, conv_factor = 9./5.)
```

2.359 PAUSE Statement (deleted feature)

Description

The PAUSE statement temporarily suspends execution of the program.

Syntax

```
PAUSE [ stop-code ]
```

Where:

stop-code is

```
scalar-char-constant                          or
digit [ digit [ digit [ digit [ digit ] ] ] ]
```

scalar-char-constant is a scalar default CHARACTER constant.

digit is a digit.

Remarks

When a PAUSE statement is reached, the diagnostic message is written to the standard error file. If the message is sent to terminal and the standard input is from the terminal, execution of the program is suspended awaiting a response. Entering any standard input data at the terminal restarts the program. In cases other than the above, the PAUSE statement is ignored and the program continues.

Example

```
pause "pausing"
```

2.360 PERROR Service Subroutine

Description

Sends a message to the standard error file.

Syntax

```
CALL  PERROR ( string )
```

Argument(s)

string

Default CHARACTER scalar. Message to go to the standard error file.

Example

```
use service_routines,only:perror
open(10,file='x.dat',err=10)
write(10,*,err=10) 'Fortran program'
```

```
   stop
10 call perror('Fortran i/o error')
   stop 200
   end
```

2.361 POINTER Statement

Description

The POINTER statement specifies a list of objects or procedure components that have the POINTER attribute.

Syntax

```
POINTER [ :: ] pointer-dcl-list
```

pointer-dcl-list is a list of pointer declarations that is the following syntax:

```
object-name [ ( deferred-shape-spec-list ) ]        or
procedure-entity-name
```

Where:

object-name is the name of an object that has the POINTER attribute.

deferred-shape-spec-list is a comma-separated list of

:

Remarks

A pointer is a data pointer or a procedure pointer.

A pointer shall not be referenced or defined unless it is first associated with a target through a pointer assignment or an ALLOCATE statement.

If a data pointer is associated and the pointer has deferred type parameters, the value becomes the type parameter value corresponding to the target.

A procedure pointer cannot be referenced unless it is pointer associated with a target procedure.

If the DIMENSION attribute is specified elsewhere in the scoping unit, the array shall have a deferred shape.

The PARAMETER attribute shall not be specified for *object-name*.

procedure-entity-name is the procedure entity name, and must be declared with an EXTERNAL attribute explicitly specified.

Example

```
real :: next, previous, value
pointer :: next, previous
```

2.362 POINTER Statement (CRAY Pointer)

The CRAY pointer specifies a pair of variables. One is a pointer variable that stores an address value, and the other is a pointer-based variable that is pointed to by the address value.

Syntax

```
POINTER (ptr,var) [,(ptr,var)] ...
```

Where:

ptr is a pointer variable eight-byte INTEGER implicitly. *ptr* must be a scalar variable. *ptr* may not be specified in a type declaration statement.

var is a pointer-based variable and can be of any type. *var* shall not be a name of the dummy argument, common block object, equivalence object, saved object, namelist group object, function result, and allocatable variable. *var* shall not specify for DO variable and scalar variable in ASSIGN statement.

Remarks

The CRAY POINTER statement shall not be specified in specification part of a module.

The CRAY POINTER statement specifies that the memory pointed to by the pointer variable *ptr* is referenced by the name of the pointer based variable *var*, keeping its type and attributes. The *var* may not be specified as other pointer variable. The *var* may be an array, it shall be an explicit shape array.

To place an actual address value in a pointer variable, use the intrinsic function LOC (see "2.310 LOC Intrinsic Function") or the service function MALLOC (see "2.323 MALLOC Service Function"). The memory obtained by the MALLOC function may be freed by the service subroutine FREE (see "2.183 FREE Service Subroutine").

Example

```
real, dimension(10) :: var
real, dimension(20) :: array
pointer (ptr,var)
ptr = loc(array)            ! Address of array is set in ptr
var(1) = 1.0                ! The reference to var is equivalent to
print *,array(1)            ! the reference to array.
end
```

2.363 Pointer Assignment Statement

Description

The pointer assignment statement causes a pointer to become associated with a target or causes its pointer association status to become disassociated or undefined. Any previous associations between a pointer and a target are deallocated. The pointer assignment statement can be for a pointer assignment or a procedure pointer assignment, and has the following format:

Syntax

```
data-pointer-object [ ( bounds-spec-list ) ] => data-target         or
data-pointer-object ( bounds-remapping-list ) => data-target        or
proc-pointer-object => proc-target
```

Where:

data-pointer-object is a data pointer object that shall have the POINTER attribute, and has the following format:

```
variable-name            or
variable % data-pointer-component-name
```

bounds-spec-list is a bounds specification list, and has the following format:

```
lower-bound-expr :
```

bounds-remapping-list is a bounds remapping list, and has the following format:

```
lower-bound-expr : upper-bound-expr
```

data-target is a data target, and has the following format:

```
variable        or
expr
```

variable is a variable that has the TARGET attribute or is a subobject with the TARGET attribute, or it has the POINTER attribute. The variable must not be a coindexed object.

expr is an expression that must return the result to be used as the data pointer.

Remarks

data-target shall be of the same type, kind type parameters, and rank as *pointer-object*.

data-target shall not be an array section with a vector subscript.

proc-pointer-objet is a procedure pointer object, and has the following format:

```
proc-pointer-name        or
proc-component-ref
```

proc-component-ref is a procedure component reference, and has the following format:

```
variable % procedure-component-name
```

A procedure pointer object must be a procedure pointer.

proc-target is a procedure target.

An elemental procedure name that is not intrinsic cannot be specified as a procedure target.

A procedure target has the following format:

```
expr                  or
procedure-name        or
proc-component-ref
```

expr is an expression. It is a function reference and must return the result to be used as the procedure pointer.

procedure-name is a procedure name.

The procedure name must be an external procedure name, a module procedure name, a dummy procedure name, a specific intrinsic function name, or a procedure pointer name.

In this case, a specific intrinsic function name must be one of the following:

```
ABS, ACOS, AIMAG, AINT, ALOG, ALOG10, AMOD, ANINT, ASIN, ATAN, ATAN2, CABS, CCOS, CEXP, CLOG,
CONJG, COS, COSH, CSIN, CSQRT, DABS, DACOS, DASIN, DATAN, DATAN2, DCOS, DCOSH, DDIM, DEXP, DIM,
DINT, DLOG, DLOG10, DMOD, DNINT, DPROD, DSIGN, DSIN, DSINH, DSQRT, DTAN, DTANH, EXP, IABS, IDIM,
IDNINT, INDEX, ISIGN, LEN, MOD, NINT, SIGN, SIN, SINH, SQRT, TAN and TANH.
```

If target is not a pointer, pointer-object becomes associated with target. If target is a pointer that is associated, pointer-object becomes associated with the same object as target. If target is a pointer that is disassociated or a reference to the NULL intrinsic function, pointer-object becomes disassociated. If target's association status is undefined, pointer's also becomes undefined.

Any previous association between pointer-object and a target is broken.

Pointer assignment for a pointer component of a structure can also take place by derived type intrinsic assignment or by a defined assignment.

A pointer also becomes associated with a target through allocation of the data pointer object by ALLOCATE statement.

A pointer shall not be referenced or defined unless it is associated with a target that may be referenced or defined.

2.363.1 Data Pointer Assignment

If a data pointer object is not polymorphic and the data target is polymorphic, and if the dynamic type differs from the declared type, the assignment target becomes a component with the data pointer object type. In other cases, the assignment target becomes the data target.

If the data target is not a pointer, the data pointer object becomes the pointer associated with the assignment target. If the data target is a pointer, the pointer association status of the data pointer object becomes the data target item, and if the data target has been associated with an object, the data pointer object becomes the item associated with the assignment target. If the data target is an allocate object, it must already be allocated.

If the data pointer object is polymorphic, the dynamic type of the data pointer object becomes the dynamic type of the target data.

If the data pointer object is the sequential derived type or is a type with the BIND attribute, the dynamic type of the target data must be that derived type.

If the pointer object has the CONTIGUOUS attribute(see "2.82 CONTIGUOUS Statement"), the pointer target must be contiguous (see "2.82 CONTIGUOUS Statement").

If the data target is an empty pointer:

Type parameters that are not deferred type parameters of the declared type of the data pointer object that correspond with similarly non-deferred type parameters of the data target must have the same value as the type parameters corresponding to the corresponding data target.

In other cases, all type parameters that are not deferred type parameters of the declared type of the data pointer object must have the same value as the type parameters corresponding to the data target.

If the data pointer object has non-deferred type parameters corresponding to deferred type parameters of the data target, the data target cannot be a pointer with indeterminate associate status.

If a bound remapping list has been specified, the data pointer cannot be an empty or indeterminate pointer, and the data target size cannot be smaller than the data pointer object size. In addition, the pointer target must be simply contiguous(see "2.82.1 Simply CONTIGUOUS") or of rank one.

If a bound remapping list is not specified, the dimension size of the data pointer object becomes the corresponding dimension size of the data target. In addition, each upper dimension bound becomes the value that is just one less than the sum of the lower bound and the size.

If a bound remapping list and a bound specification list are not specified, the minimum for each dimension is the value of the result of the intrinsic function LBOUND when the data target is the argument.

In other cases, the each lower dimension is the result for the dimension of the data target corresponding to the intrinsic function LBOUND.

Each upper dimension bound becomes the sum of the lower bound and the size minus one.

Example

```
real, pointer :: a
real, target :: b = 5.0
a => b               ! a become associated with b
end
```

2.363.2 Procedure Pointer Assignment

If a procedure target is not a pointer, the procedure pointer object is in contention with the procedure target. If it is a pointer, the pointer association status of the procedure pointer object becomes the procedure target and, when the procedure target is associated with the procedure, the procedure pointer object is associated with the same procedure.

If the procedure pointer object has an explicit interface, its characteristics must be the same as the procedure target. However, if the procedure target is pure and the procedure pointer object is not pure, or if the procedure target is an elemental intrinsic procedure and the procedure pointer object is not, the characteristics need not be the same.

If the characteristics of the procedure pointer object or the procedure target require an explicit interface, the procedure pointer object and procedure target must both have an explicit interface.

If the procedure pointer object has an implicit interface and it is referenced as an explicit type or a function, the procedure target must be a function.

If the procedure pointer object has an implicit interface and it is referenced as a subroutine, the procedure target must be a subroutine.

If the procedure target and the procedure pointer object are functions, the functions must be the same type, and corresponding type parameters must both be unspecified or both have the same value.

If the procedure name is a specific procedure name and a generic name, the specific procedure is associated with the pointer object.

Example

```
interface
 function ifunc () result(irst)
  integer :: irst
 end function
end interface
procedure(ifunc), pointer :: pifunc
pifunc => ifunc ! The procedure pointer pifunc will be
                ! associated with the procedure ifunc.
print *,pifunc()
end
function ifunc() result(irst)
 integer :: irst
```

```
 irst=1
end function
```

2.364 POPCNT Intrinsic Function

Description

Number of 1 bits.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
POPCNT	---	1	INTEGER	Default INTEGER

```
result = POPCNT( I )
```

Argument(s)

I

I shall be of type INTEGER.

Result

The result is of the default INTEGER type.

Remarks

POPCNT evaluates the number of 1 bits in the sequence of bits of *I*.

The result is of the default INTEGER type.

Example

```
k = popcnt(9)    ! k is assigned the value 2
```

2.365 POPPAR Intrinsic Function

Description

Parity expressed as 0 or 1.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
POPPAR	---	1	INTEGER	Default INTEGER

```
result = POPPAR( I )
```

Argument(s)

I

I shall be of type INTEGER.

Result

The result is of the default INTEGER type.

Remarks

The result is the value 1 if POPCNT(*I*) is odd. The result is the value 0 if POPCNT(*I*) is even.

The result is of the default INTEGER type.

Example

```
k = poppar(2)    ! k is assigned the value 1
n = poppar(3)    ! n is assigned the value 0
```

2.366 PRECFILL Service Subroutine

Description

Set fill character for numeric fields those are wider than supplied numeric precision.

Syntax

```
CALL PRECFILL ( letter )
```

Argument(s)

letter

1 byte default CHARACTER scalar. The new precision fill character.

Example

```
use service_routines,only:precfill
write(*,'(d24.17)') 1.0d+01
CALL precfill('*')
write(*,'(d24.17)') 1.0d+01
end
```

2.367 PRECISION Intrinsic Function

Description

Decimal precision of data type.

Class

Inquiry function.

Syntax

```
result = PRECISION ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX. It can be scalar or array-valued.

Result

The result is of type default INTEGER. Its value is equal to the number of decimal digits of precision in the data type of *X*.

The result value is as follows:

Type of *X*	The result value
Single precision REAL	6
Double precision REAL	15
Quadruple precision REAL	33
Single precision COMPLEX	6
Double precision COMPLEX	15
Quadruple precision COMPLEX	33

Example

```
i = precision (4.2) ! i is assigned the value 6
```

2.368 PRESENT Intrinsic Function

Description

Determine whether an optional argument is present.

Class

Inquiry function.

Syntax

```
result = PRESENT ( A )
```

Argument(s)

A

A shall be an optional argument of the procedure in which the PRESENT function appears. It can be any type, including scalar, an array, or a dummy procedure. It does not have the INTENT attribute.

Result

The result is a scalar default LOGICAL. Its value is true if the actual argument corresponding to *A* was provided in the invocation of the procedure in which the PRESENT function appears and false otherwise.

Example

```
call zee(a,b)
contains
  subroutine zee (x,y,z)
    implicit none
    real, intent(inout) :: x, y
    real, intent(in), optional :: z
    logical r
    r = present(z) ! r is assigned the value false
    ...
```

2.369 PRINT Statement

Description

The PRINT statement transfers values from an output list and format specification to a file.

Syntax

```
PRINT format [ , output-item-list ]
```

Where:

format is

```
default-char-expr                or
label                            or
*                                or
scalar-default-int-variable
```

default-char-expr is a default CHARACTER expression that evaluates to *format-specification*. See "1.8.1 Format Specification" for format specification.

label is a statement label of a FORMAT statement in the same scoping unit as the PRINT statement.

scalar-default-int-variable is a scalar default INTEGER variable that was assigned the label only by an ASSIGN statement of a FORMAT statement in the same scoping unit.

output-item-list is a comma-separated list of

```
expr                or
io-implied-do
```

expr is an expression.

io-implied-do is

```
( output-item-list , io-implied-do-control )
```

io-implied-do-control is

```
do-variable = scalar-expr , scalar-expr [ , scalar-expr ]
```

do-variable is a named scalar variable of type INTEGER, default REAL, or double precision REAL. The *do-variable* of type default REAL or double precision REAL is deleted feature.

scalar-expr is a scalar expression of type INTEGER, default REAL, or double precision REAL. The *scalar-expr* is of type default REAL or double precision REAL is deleted feature.

Remarks

For an implied-DO, the loop initialization and execution is the same as for a DO construct (see "2.114 DO Construct").

The *do-variable* of an *io-implied-do-control* that is contained within another *io-implied-do* shall not appear as the *do-variable* of the containing *io-implied-do*.

If an output item is a pointer, it shall be currently associated with a target and data are transferred from the target to the file.

If an output item is an allocatable variable, it shall be currently allocated.

If an array appears as an output item, it is treated as if the elements are specified in array-element order (see "1.5.8.3 Array Element Order").

If a derived type ultimately contains a pointer component or an allocatable component, an object of this type shall not appear as the result of the evaluation of an output list item.

If a derived type object appears as an output item, it is treated as if all of the components are specified in the same order as in the definition of the derived type.

Example

```
        print*,"hello world"
        print 100, i,j,k
100     format (3i8)
```

2.370 PRIVATE Statement

Description

The PRIVATE statement specifies that the names of entities are accessible only within the current module. The PRIVATE statement is permitted only in a module.

Syntax

```
PRIVATE [ [ :: ] access-id-list ]
```

Where:

access-id-list is a comma-separated list of

```
use-name          or
generic-spec
```

use-name is the name of a named variable, procedure, derived type, named constant, or namelist group.

generic-spec is

```
generic-name                    or
OPERATOR ( defined-operator )   or
```

```
ASSIGNMENT ( = )                  or
dtio-generic-spec
```

generic-name is the name of a generic procedure.

defined-operator is

```
intrinsic-operator        or
. operator-name .
```

operator-name is a user-defined name for the operation, consisting of one to 240 letters.

ASSIGNMENT (=) is a user-defined assignment.

dtio-generic-spec is derived type input/output procedure that is the following syntax:

```
READ ( FORMATTED )        or
READ ( UNFORMATTED )      or
WRITE ( FORMATTED )       or
WRITE ( UNFORMATTED )
```

Remarks

If the PRIVATE statement appears without a list of objects, it sets the default accessibility of named items in the module to private. Otherwise, the default is public accessibility. If the PRIVATE statement appears with *access-id-list*, it makes the accessibility of the objects specified private.

The PRIVATE statement may appear in a derived type definition in a module. If the PRIVATE statement appears in a derived type definition, the entities within the derived type definition are accessible only in the current module. Similarly, the structure constructor for such a type shall be employed only within the defining module. Within a derived type definition, the PRIVATE statement shall not appear with a list of *access-id-list*. See "1.5.11.1 Derived Type Definition" for derived type definition.

Example

```
module ex
  implicit none
  public     ! default accessibility is public
  real :: a, b, c
  private a  ! a is not accessible outside module
             ! b and c are accessible outside module
  type zee
    private         ! the component and structure constructor are
    integer :: l,m  ! not accessible outside module
  end type zee
end module ex
```

2.371 PRNSET Service Subroutine

Description

Changes the specified value for precision lowering function.

Syntax

```
CALL  PRNSET ( i )
```

Argument(s)

i

Default INTEGER scalar. The value shall be 0 to 15.

Remarks

If you use PRNSET, the precision lowering function shall be effective. If it is not effective, PRNSET is ignored. See "Fortran User's Guide" for precision lowering function.

Example

```
use service_routines,only:prnset
real :: r4
call prnset(0)
r4 = -1.234567
write(1,*) r4
call prnset(1)
write(1,*) r4
end
```

2.372 PROCEDURE Statement

Description

The PROCEDURE statement is

- Procedure component definition statement (see "1.5.11.1 Derived Type Definition")
- Specific binding (see "1.5.11.3 Type-Bound Procedure")
- Interface block
- Procedure declaration statement (see "2.373 Procedure Declaration Statement")

The PROCEDURE statement in an interface block specifies that the names in the *procedure-name-list* are part of a generic interface.

Syntax

```
[ MODULE ] PROCEDURE procedure-name-list
```

Where:

procedure-name-list is a comma-separated list of module procedures accessible by host or use association.

The PROCEDURE statement can be specified only in an interface block that has a generic specifier.

If MODULE is specified in a PROCEDURE statement, each procedure name in that statement must be accessible as a module procedure within the scope.

A procedure declared by an earlier PROCEDURE statement in an accessible interface that has the same generic identifier cannot be specified for a procedure name.

Remarks

See "1.12.7.2 Procedure Interface Block" for interface block.

Example

```
module names
  interface bill
    module procedure fred, jim
  end interface
  contains
  function fred(i)
    integer :: i
    ...
  end function fred
  function jim(r)
    real :: r
    ...
  end function jim
end module names
```

2.373 Procedure Declaration Statement

Description

The procedure declaration statement declares a procedure pointer, a dummy procedure, and external procedure with EXTERNAL attribute.

Syntax

```
PROCEDURE ([ proc-interface ])[[,proc-attr-spec ] ... ::]proc-decl-list
```

Where:

proc-interface is

```
interface-name              or
declaration-type-spec
```

proc-attr-spec is

```
access-spec                     or
proc-language-binding-spec      or
INTENT ( intent-spec )          or
OPTIONAL                        or
POINTER                         or
SAVE
```

proc-language-binding-spec is

```
BIND ( C [, NAME = scalar-char-initialization-expr ] )
```

proc-decl-list is

```
procedure-entity-name[ => null-init]
```

Remarks

null-init must be a reference to an intrinsic function NULL with no arguments.

The interface name must be the name of an abstract interface or a procedure with an explicit interface.

If the interface name is declared in a procedure declaration statement, it must be declared in advance. If the interface name indicates an intrinsic procedure, it must be one of the following:

```
ABS, ACOS, AIMAG, AINT, ALOG, ALOG10, AMOD, ANINT, ASIN, ATAN, ATAN2, CABS, CCOS, CEXP, CLOG,
CONJG, COS, COSH, CSIN, CSQRT, DABS, DACOS, DASIN, DATAN, DATAN2, DCOS, DCOSH, DDIM, DEXP, DIM,
DINT, DLOG, DLOG10, DMOD, DNINT, DPROD, DSIGN, DSIN, DSINH, DSQRT, DTAN, DTANH, EXP, IABS, IDIM,
IDNINT, INDEX, ISIGN, LEN, MOD, NINT, SIGN, SIN, SINH, SQRT, TAN, or TANH.
```

The interface name cannot be the same as a keyword that declares an intrinsic type.

If the INTENT attribute or the SAVE attribute is specified, the POINTER attribute must also be specified.

If there is an elemental procedure specification in the procedure interface, those procedure element names must specify external procedures.

If => is specified in a procedure declaration, the POINTER attribute must be specified.

If a NAME= specifier specifies a procedure language binding specifier, only one procedure declaration must be specified in the procedure language list. The procedure declaration cannot specify the POINTER attribute and cannot be a dummy procedure.

If a procedure language binding specifier is specified, the procedure interface must be specified and must be an interface name declared with a procedure language binding specifier.

If a procedure interface is specified and is from an interface name, the procedure declared or the explicit specific interface corresponding to the procedure pointer is declared. The abstract interface is specified by interface name.

If a procedure interface is specified and is from a declaration type specifier, the declared procedure or procedure pointer is a function with an implicit interface and the specified result type. If the type is specified in an external procedure, the same result type and kind type parameter must be specified in that function definition.

If a procedure interface is not specified, the procedure declaration statement does not indicate whether the declared procedure or procedure pointer is a subroutine or a function.

If '=>null-init' is specified in the procedure declaration of a procedure declaration statement, the initial associate status of the corresponding procedure element is disassociated. This signifies that it has the SAVE attribute. The SAVE attribute can also be explicitly specified in the procedure declaration statement or declared again using the SAVE statement.

2.374 PRODUCT Intrinsic Function

Description

Product of elements of an array, along a given dimension, for which a mask is true.

Class

Transformational function.

Syntax

```
result = PRODUCT ( ARRAY [ , MASK ] )          or
result = PRODUCT ( ARRAY , DIM [ , MASK ] )
```

Required Argument(s)

ARRAY

ARRAY shall be of type INTEGER, REAL or COMPLEX. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar INTEGER in the range 1 <= *DIM* <= *n*, where *n* is the rank of *ARRAY*. The corresponding dummy argument shall not be an optional dummy argument.

MASK

MASK shall be of type LOGICAL and shall be conformable with *ARRAY*.

Result

The result is of the same type and kind as *ARRAY*. It is scalar if *DIM* is absent or if *ARRAY* has rank one; otherwise the result is an array of rank *n*-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of *ARRAY*. If *DIM* is absent, the value of the result is the product of the values of all the elements of *ARRAY*. If *DIM* is present, the value of the result is the product of the values of all elements of *ARRAY* along dimension *DIM*. If *MASK* is present, the elements of *ARRAY* for which *MASK* is false are not considered.

Example

```
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
integer j(2)
! m is the array |1 3|
!                |2 4|
i = product(m)          ! i is assigned the value 24
j = product(m,dim=1)    ! j is assigned the value [2,12]
k = product(m,mask=m>2) ! k is assigned the value 12
```

2.375 PROGRAM Statement

Description

The PROGRAM statement begins a main program. See "1.11.1 Main Program" for main program.

Syntax

```
PROGRAM program-name
```

Where:

program-name is the name of the main program.

Remarks

program-name is global entities of a program, it shall not be used to identify other program units, external procedures, and common blocks in the same program. Also it shall not be used to identify a local entity in the main program.

Example

```
program main
  ...
end program main
```

2.376 PROMPT Service Subroutine

Description

Sets and outputs the prompt for sequential read statements using the standard input file. The prompt service subroutine is effective only when it is used on TSS.

Syntax

```
CALL  PROMPT ( string )
```

Argument(s)

string

Default CHARACTER scalar. Prompt message.
If the length of string is longer than 151, PROMPT truncates the argument.

Example

```
use service_routines,only:prompt
read(*,*) i
call prompt('>>>')
read(*,*) i
end
```

2.377 PROTECTED Statement

Description

The PROTECTED statement specifies that the named variable is specified in a MODULE program unit, the part in the useful range by use association where the variable can be modified is limited.

Syntax

```
PROTECTED [ :: ] entity-name-list
```

Where:

entity-name-list is a comma-separated list of an entity name.

The PROTECTED statement is permitted only in the specification part of a module.

Example

```
module mod
  integer :: i, j, k
  protected :: i   ! The PROTECTED attribute is specified to "i",
                   ! so "i" cannot be changed at any place excluding
                   ! an effective area in module mod.
end module
```

2.378 PUBLIC Statement

Description

The PUBLIC statement specifies that the names of entities are accessible anywhere the module in which the PUBLIC statement appears is used. The PUBLIC statement is permitted only in a module.

Syntax

```
PUBLIC [ [ :: ] access-id-list ]
```

Where:

access-id-list is a comma-separated list of

```
use-name        or
generic-spec
```

use-name is the name of a named variable, procedure, derived type, named constant, or namelist group.

generic-spec is

```
generic-name                    or
OPERATOR ( defined-operator )   or
ASSIGNMENT ( = )                or
dtio-generic-spec
```

generic-name is the name of a generic procedure.

defined-operator is

```
intrinsic-operator      or
. operator-name .
```

operator-name is a user-defined name for the operation, consisting of one to 240 letters.

ASSIGNMENT (=) is a user-defined assignment.

dtio-generic-spec is a derived type input/output procedure that is the following syntax:

```
READ ( FORMATTED )      or
READ ( UNFORMATTED )    or
WRITE ( FORMATTED )     or
WRITE ( UNFORMATTED )
```

Remarks

The default accessibility of names in a module is public if the PRIVATE statement does not appear without a list of objects. If the PUBLIC statement appears without a list of objects, it confirms the default accessibility. If a list of objects is present, the PUBLIC statement makes the accessibility of the objects specified public.

Example

```
module zee
  implicit none
  private   ! default accessibility is now private
  real :: a, b, c
  public a  ! a is now accessible outside module
            ! b and c are not accessible outside module
end module zee
```

2.379 PUTC Service Function

Description

Writes a character to the standard output file.

Syntax

```
iy = PUTC ( ch )
```

Argument(s)

ch

Default CHARACTER(1) scalar. A character to be written to the standard output file.

Result

Default INTEGER scalar. The value of each function is zero if successful, and -1 or a Fortran runtime message number otherwise.

Example

```
use service_routines,only:putc
integer :: x
character(len=6) :: pr='input>'
do i=1,6
  if (putc(pr(i:i)) .ne. 0) stop 10
end do
read(*,*) x
end
```

2.380 QEXT Intrinsic Function

Description

Convert to quadruple precision REAL type.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
QEXT	---	1	One-byte INTEGER	Quad-prec REAL
	---		Two-byte INTEGER	Quad-prec REAL
	QFLOAT		Four-byte INTEGER	Quad-prec REAL
	---		Eight-byte INTEGER	Quad-prec REAL
	QEXT		Single-prec REAL	Quad-prec REAL
	QEXTD		Double-prec REAL	Quad-prec REAL
	---		Quad-prec REAL	Quad-prec REAL
	---		Single-prec COMPLEX	Quad-prec REAL
	---		Double-prec COMPLEX	Quad-prec REAL
	QREAL		Quad-prec COMPLEX	Quad-prec REAL

```
result = QEXT ( A )
```

Argument(s)

A

A shall be of type INTEGER, REAL or COMPLEX.

Result

The result is of quadruple precision REAL type. Its value is a quadruple precision representation of *A*. If *A* is of type COMPLEX, the result is a quadruple precision representation of the real part of *A*.

Remarks

QEXT, QFLOAT, QEXTD and QREAL convert to quadruple precision REAL type. If *A* is of type COMPLEX, the result is as much precision of the significant part of the real part of *A* as a quadruple precision REAL datum can contain.

The generic name, QEXT, may be used with any INTEGER, REAL, or COMPLEX argument.

The type of the result of each function is quadruple precision real.

Example

```
real(16) :: q
q = qext(2.0)  ! q is assigned the value 2.0q0
```

2.381 QPROD Intrinsic Function

Description

Quadruple precision REAL product.

Class

Elemental function.

Syntax

```
result = QPROD ( X , Y )
```

Argument(s)

X

X shall be of type double precision REAL.

Y

Y shall be of type double precision REAL.

Result

The result is of type quadruple precision REAL. Its value is a quadruple precision representation of the product of *X* and *Y*.

Example

```
real( 8) :: a,b
real(16) :: c
a = 2.0d0
b = 3.0d0
c = qprod(a,b)     ! c is assigned the value 6.0q0
```

2.382 QSORT Service Subroutine

Description

Performs a quick sort on an array of rank one.

Syntax

```
CALL  QSORT ( array , nel , width , compare )
```

Argument(s)

array

Any type. One-dimensional array to be sorted.

nel

Default INTEGER scalar. *array*'s size.

width

Default INTEGER scalar. Size, in bytes, of a single element of *array*:

4 if *array* is of type single precision REAL
8 if *array* is double precision REAL or single precision COMPLEX
16 if *array* is double precision COMPLEX

compare

Two-byte INTEGER. Name of a user-defined ordering function that determines sort order.

The type declaration of compare takes the form:

```
INTEGER(2)  FUNCTION compare(arg1, arg2)
```

where *arg1* and *arg2* have the same type as array. Once you have created an ordering scheme, implement your sorting function so that it returns the following:

Negative if *arg1* should precede *arg2*
Zero if *arg1* is equivalent to *arg2*
Positive if *arg1* should follow *arg2*

Example

```
! SORT
   use service_routines,only:qsort
   integer(4),dimension(10) :: array
   integer(2),external :: compare4
   array=(/4,6,8,2,10,5,3,1,7,9/)
   write(6,*) array               ! Outputs 4 6 8 2 10 5 3 1 7 9
   call qsort(array,10,4,compare4)
   write(6,*) array               ! Outputs 1 2 3 4 5 6 7 8 9 10
   end

   integer(2) function compare4(i1,i2)
   integer(4) i1,i2
   if(i1-i2) 10,20,30
10 compare4 = -1
   return
20 compare4 = 0
   return
30 compare4 = 1
   return
   end
```

2.383 RADIX Intrinsic Function

Description

Number base of the physical representation of a number.

Class

Inquiry function.

Syntax

```
result = RADIX ( X )
```

Argument(s)

X

X shall be of type INTEGER or REAL. It can be scalar or array-valued.

Result

The result is a default INTEGER scalar whose value is the number base of the physical representation of *X*. Always it has a value 2.

Example

```
i = radix(2.3)  ! i is assigned the value 2
```

2.384 RAN Service Function

Description

Generates a random number greater than or equal to 0.0 and less than or equal to 1.0.

Syntax

```
y = RAN ( seed )
```

Argument(s)

seed

Default INTEGER scalar. Seed for the random number generator.

Result

Single precision REAL, scalar. Pseudo random number, x, where $0.0 < x < 1.0$.

Example

```
use service_routines,only:ran
real(4) :: x(10)
integer(4) :: seed=123456
do i=1,10
  x(i)=ran(seed)           ! Generates 10 random numbers
end do
end
```

2.385 RAND Service Function

Description

Generates a random number greater than or equal to 0.0 and less than or equal to 1.0.

Syntax

```
y = RAND ( i )
```

Argument(s)

i

Default INTEGER scalar.

Result

Single precision REAL, scalar. The values are as follows:

Argument(*i*)	Selection process
0	The next random number in the sequence is selected.
1	The generator is restarted and the first random value is selected.
Otherwise	The generator is reseeded using *i*, restarted, and the first random value is selected.

Example

```
use service_routines,only:rand
do i=1,10
  print *,rand(0)     ! Generates 10 random numbers
end do
end
```

2.386 RANDOM_NUMBER Intrinsic Subroutine

Description

Uniformly distributed pseudorandom number or numbers in the range 0 <= *x* < 1. The generator uses a multiplicative congruential algorithm with a period of approximately 2^{38}.

Class

Subroutine.

Syntax

```
CALL RANDOM_NUMBER ( HARVEST )
```

Argument(s)

HARVEST

HARVEST shall be of type REAL. It is an INTENT(OUT) argument. It can be a scalar or an array variable. Its value is one or several pseudorandom numbers uniformly distributed in the range 0 <= *HARVEST* < 1.

Example

```
real, dimension(8) :: x
call random_number(x) ! each element of x is assigned
                      ! a pseudorandom number
```

2.387 RANDOM_SEED Intrinsic Subroutine

Description

Set or query the pseudorandom number generator used by RANDOM_NUMBER. If no argument is present, the processor sets the seed to a predetermined value.

Class

Subroutine.

Syntax

```
CALL RANDOM_SEED ( [ SIZE , PUT , GET ] )
```

Optional Argument(s)

SIZE

SIZE shall be a scalar of type default INTEGER. It is an INTENT(OUT) variable. It is set to the number of default INTEGERs the processor uses to hold the seed. The value of the number in this system is 1.

PUT

PUT shall be a default INTEGER array of rank one and size greater than or equal to *SIZE*. It is an INTENT(IN) argument and is used by the processor to set the seed value.

GET

GET shall be a default INTEGER array of rank one and size greater than or equal to *SIZE*. It is an INTENT(OUT) argument and is set by the processor to the current value of the seed.

Remarks

Exactly one or zero arguments shall be present.

If no argument is present, the processor initializes the seed.

Example

```
integer, dimension(100) :: seed,old
call random_seed                ! initialize the seed
call random_seed(size=k)        ! k set to size of seed
```

```
call random_seed(put=seed(1:k))  ! set user seed
call random_seed(get=old(1:k))   ! get current seed
```

2.388 RANGE Intrinsic Function

Description

Decimal range of the data type of a number.

Class

Elemental function.

Syntax

```
result = RANGE ( X )
```

Argument(s)

X

X shall be of numeric type. It can be scalar or array-valued.

Result

The result is a scalar default INTEGER. If *X* is of type INTEGER, the result value is INT(LOG10(HUGE(*X*))). If *X* is of type REAL or COMPLEX, the result value is INT(MIN(LOG10(HUGE(*X*)),-LOG10(TINY(*X*)))).

The result value is as follows:

Type of *X*	The result value
One-byte INTEGER	2
Two-byte INTEGER	4
Four-byte INTEGER	9
Eight-byte INTEGER	18
Single precision REAL	37
Double precision REAL	307
Quadruple precision REAL	4931
Single precision COMPLEX	37
Double precision COMPLEX	307
Quadruple precision COMPLEX	4931

Example

```
i = range(4.2)   ! i is assigned the value 37
```

2.389 READ Statement

Description

The READ statement transfers values from a file to the entities specified in an input list or a namelist group.

Syntax

```
READ ( io-control-spec-list ) [ input-item-list ]      or
READ format [ , input-item-list ]
```

Where:

io-control-spec-list is a comma-separated list of

```
[ UNIT = ] io-unit                or
[ FMT = ] format                  or
[ NML = ] namelist-group-name     or
REC = record-number               or
IOSTAT = io-stat                  or
ERR = err-label                   or
END = end-label                   or
ADVANCE = advance                 or
SIZE = size                       or
EOR = eor-label                   or
NUM = record-len                  or
ASYNCHRONOUS = asynchronous       or
BLANK = blank                     or
DECIMAL = decimal                 or
ID = id                           or
IOMSG = iomsg                     or
PAD = pad                         or
POS = pos                         or
ROUND = round
```

An *io-unit* shall be specified. If the optional characters UNIT= are omitted from the *io-unit* specifier, the *io-unit* specifier shall be the first item in the *io-control-spec-list*.

io-unit is

```
external-file-unit        or
*                         or
internal-file-unit
```

external-file-unit is a scalar INTEGER expression that evaluates to the input/output unit number of an external file.

internal-file-unit is a default CHARACTER variable that is to the input/output unit file of an internal file.

If the *io-unit* specifier specifies an internal file, the REC= specifier or POS= specifier shall not be specified.

If the *io-unit* specifier specifies an asterisk, the *io-unit* specifier effect on image one.

io-control-spec-list shall not contain both a *format* and a *namelist-group-name*.

If the optional characters FMT= are omitted from the format specifier, the *format* specifier shall be the second item in the *io-control-spec-list* and the first item shall be the *io-unit* specifier without the optional characters UNIT=.

format is

```
default-char-expr               or
label                           or
*                               or
scalar-default-int-variable
```

default-char-expr is a default CHARACTER expression that evaluates to *format-specification*. See "1.8.1 Format Specification" for format specification.

label is a statement label of a FORMAT statement in the same scoping unit as the READ statement.

scalar-default-int-variable is a scalar default INTEGER variable that was assigned the label only by an ASSIGN statement of a FORMAT statement in the same scoping unit.

If the optional characters NML= are omitted from the *namelist-group-name* specifier, the *namelist-group-name* specifier shall be the second item in the *io-control-spec-list* and the first item shall be the *io-unit* specifier without the optional characters UNIT=.

namelist-group-name is the name of a namelist group.

If the *namelist-group-name* is present, the *input-item-list* shall not be present.

record-number is a scalar INTEGER expression that is to the number of the direct access record that is to be read.

If the REC= specifier is present, an END= specifier, a POS= specifier, and a NML= specifier shall not appear, a *namelist-group-name* shall not appear, and *format* shall not be an asterisk indicating list-directed I/O.

io-stat is a scalar default INTEGER variable that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, -1 if an end-of-file condition occurs, -2 if an end-of-record condition occurs, and zero otherwise.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the READ statement. If an error condition occurs during execution of the READ statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

end-label is a statement label of a branch target statement that appears in the same scoping unit as the READ statement. If an end-of-file condition occurs during execution of the READ statement that contains an END= specifier, execution continues with the statement specified in the END= specifier.

advance is a scalar default CHARACTER expression that evaluates to NO if non-advancing input/output is to occur, and YES if advancing input/output is to occur. The default value is YES.

An ADVANCE= specifier may be present only in a formatted sequential input/output statement with explicit format specification whose control information list does not contain an internal file *io-unit* specifier.

size is a scalar default INTEGER variable that is assigned the number of characters transferred by data edit descriptors during execution of the current non-advancing input/output statement.

A SIZE= specifier may be present only in an input statement that contains an ADVANCE= specifier with the value NO.

eor-label is a statement label of a branch target statement that appears in the same scoping unit as the READ statement. If an end-of-record condition occurs during execution of the READ statement that contains an EOR= specifier, execution continues with the statement specified in the EOR= specifier.

An EOR= specifier may be present only in an input statement that contains an ADVANCE= specifier with the value NO.

record-len is a scalar INTEGER variable that is assigned the record length in units of bytes that is actually transferred by executing an unformatted input statement.

If the NUM= specifier is present, a *format* and a *namelist-group-name* shall not appear.

input-item-list is a comma-separated list of

```
variable              or
io-implied-do
```

variable is a variable. *variable* shall not be a whole assumed shape array.

io-implied-do is

```
( input-item-list , io-implied-do-control )
```

io-implied-do-control is

```
do-variable = scalar-expr , scalar-expr [ , scalar-expr ]
```

do-variable is a named scalar variable of type INTEGER, default REAL, or double precision REAL. The *do-variable* of type default REAL or double precision REAL is deleted feature.

scalar-expr is a scalar expression of type INTEGER, default REAL, or double precision REAL. The *scalar-expr* is of type default REAL or double precision REAL is deleted feature.

Remarks

For an implied-DO, the loop initialization and execution is the same as for a DO construct (see "2.114 DO Construct").

The *do-variable* of an *io-implied-do-control* that is contained within another *io-implied-do* shall not appear as the *do-variable* of the containing *io-implied-do*.

If an input item is a pointer, it shall be currently associated with a definable target and data are transferred from the file to the associated target.

If an input item is an allocatable variable, it shall be currently allocated.

If an array appears as an input item, it is treated as if the elements are specified in array-element order (see "1.5.8.3 Array Element Order").

asynchronous must be a default CHARACTER.

The ASYNCHRONOUS= specifier specifies whether an input/output statement is synchronous or asynchronous. The asynchronous value must be either 'YES' or 'NO'.

If 'YES' is specified, the statement and input/output operation are asynchronous. If 'NO' is specified or if it is omitted, the statement and input/output operation are synchronous.

If the device identifier is not a file device number, 'YES' cannot be specified. If the ID= specifier is specified, 'YES' must be specified.

Asynchronous input/output is permitted only for external files with an OPEN statement in which 'YES' is specified in the ASYNCHRONOUS= specifier.

Synchronous input/output and asynchronous input/output are both possible for files with 'YES' specified in the ASYNCHRONOUS= specifier. For other files (that is, files opened with 'NO' in the ASYNCHRONOUS= specifier, files opened with the ASYNCHRONOUS= specifier omitted, previously connected files referenced without an OPEN statement, and internal files) only synchronous input/output is possible.

For external files, records and file storage units read and written by asynchronous data transfer can be read and written in the same sequence as when the data transfer statement is synchronous.

If any of the variables below are specified in an asynchronous data transfer statement, the ASYNCHRONOUS attribute is implicitly given to the data reference objects within the valid range of the data transfer statement. This attribute can be defined by an explicit declaration.

1. Input/output item list
2. Variable group entity
3. SIZE= specifier

blank must be a scalar default CHARACTER expression. The value of *blank* is either 'NULL' or 'ZERO'. The BLANK= specifier temporarily changes the blank interpretation mode of that connection. If it is omitted, the mode is not changed.

decimal must be a scalar default CHARACTER expression. The *decimal* value must be either 'COMMA' or 'POINT'. It specifies the value of the current decimal editing mode at this connection. If this specifier is omitted , the mode is not changed.

id must be a scalar INTEGER variable. When an asynchronous data transfer statement with an ID specifier is successfully executed, the variable specified by the ID= specifier is determined. This value can be used by subsequent WAIT statements or INQUIRE statements to identify a particular data transfer operation.

If an error occurs during execution of the data transfer statement, the variable specified in the ID= specifier becomes undefined.

An ID= specifier cannot be specified in a child data transfer statement.

iomsg must be a scalar default CHARACTER variable. If the following conditions occur during input/output statement execution, an explanatory message is assigned to *iomsg*:

- Error condition
- File end condition
- Record end condition

In other cases, the *iomsg* value does not change.

pad must be a scalar default CHARACTER expression. The value of *pad* is either 'YES' or 'NO'. The PAD= specifier temporarily changes the blank supplement mode of that connection. If it is omitted, the mode is not changed. If the POS= specifier is specified, the REC= specifier cannot be specified in the data transfer specifier list.

pos must be a scalar INTEGER expression. The POS= specifier specifies the file position within a file storage unit. It can be specified in a data transfer statement only if a device connected as a stream access is specified. It cannot be specified in a child data transfer statement.

A processor can restrict the use of the POS= specifier for specific files that do not have the function required for random positioning. If the processor also does not have the function required for applying positioning to a file, it may prohibit positioning at a file position that precedes the current file position.

For example, a file connected to a data stream device or similar cannot be positioned. If a file is connected as a formatted stream access, the file position specified by the POS= specifier must be the same as 1, that is, the file start, or the value of the file position to which an INQUIRE statement POS= specifier has previously returned.

round must be a scalar default CHARACTER expression. The round value is one of the values prescribed by the OPEN statement ROUND= specifier. The ROUND= specifier temporarily changes the input/output rounding mode of that connection. If ROUND is omitted, the mode is not changed.

If the DECIMAL=, BLANK=, PAD=, or ROUND= specifier is specified, the format identifier or the variable group name must also be specified.

Example

```
read *, a,b,c     ! read into a, b, and c using list-directed i/o
read (3, fmt= "(e7.4)") x
                  ! read in x from unit 3 using e editing
read 10, i,j,k
                  ! read in i, j, and k using format at
                  ! label 10
```

2.390 REAL Intrinsic Function

Description

Convert to REAL type.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
REAL	---	1 or 2	INTEGER or REAL or COMPLEX or binary, octal, or hexadecimal constant [, INTEGER]	REAL
	---	1	One-byte INTEGER	Single-prec REAL
	FLOATI		Two-byte INTEGER	Single-prec REAL
	REAL		Four-byte INTEGER	Single-prec REAL
	FLOAT		Four-byte INTEGER	Single-prec REAL
	FLOATJ		Four-byte INTEGER	Single-prec REAL
	---		Eight-byte INTEGER	Single-prec REAL
	---		Single-prec REAL	Single-prec REAL
	SNGL		Double-prec REAL	Single-prec REAL
	SNGLQ		Quad-prec REAL	Single-prec REAL
	---		Single-prec COMPLEX	Single-prec REAL
	---		Double-prec COMPLEX	Double-prec REAL
	---		Quad-prec COMPLEX	Quad-prec REAL

result = REAL (*A* [, *KIND*])

Required Argument(s)

A

A shall be of type INTEGER, REAL, COMPLEX , or binary, octal, or hexadecimal constant.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is of type REAL. If *KIND* is specified, the kind type parameter of the function result is following to *KIND*. If *KIND* is omitted, the kind type parameter of the function result is default REAL.

The result value is a REAL representation of *A*. If *A* is of type COMPLEX, the result's value is a REAL representation of the real part of *A*. If *A* is of binary, octal, or hexadecimal constant, the result's value is same value of a REAL type variable that has bit sequence as specified the binary, octal, or hexadecimal constant.

Remarks

REAL, FLOATI, FLOAT, FLOATJ, SNGL, and SNGLQ convert to REAL type.

The generic name, REAL, may be used with any INTEGER, REAL, COMPLEX, or binary, octal, or hexadecimal constant argument.

The type of the result is real with the kind type parameter specified by *KIND* if present. Otherwise, the kind is default REAL for INTEGER or REAL arguments and the same as the kind type of the argument for COMPLEX arguments.

Example

```
b = real(-3)  ! b is assigned the value -3.0
```

2.391 REAL Type Declaration Statement

Description

The REAL type declaration statement declares entities of type REAL. See "2.469 Type Declaration Statement" for type declaration statement.

Syntax

```
REAL [kind-selector][[,attr-spec] ... ::]entity-decl-list
```

2.392 RECORD Statement

Description

The RECORD statement declares entities of derived type *type-name*.

Syntax

```
RECORD / type-name / entity-decl-list [ , / type-name / entity-decl-list ] ...
```

Where:

type-name is a derived type name. *type-name* shall have been defined previously in the scoping unit or be accessible there by use or host association.

entity-decl-list is a comma-separated list of

```
object-name [ ( array-spec ) ]or
function-name
```

array-spec is an array specification. See "2.113 DIMENSION Statement" for array specification.

object-name is the name of a data object being declared.

function-name is the name of a function being declared. *function-name* shall be the name of the external function, intrinsic function, dummy function, or statement function.

Example

```
structure /complex_element/
  union
```

```
    map
      real :: real,imag
    end map
    map
      complex :: complex
    end map
  end union
end structure
record /complex_element/ x
x%real = 2.0
x%imag = 3.0
print *,x%complex     ! complex has the value (2.0,3.0)
```

2.393 REDLEN Service Subroutine

Description

REDLEN gets the Fortran record length that is read immediately before.

Syntax

```
CALL  REDLEN ( i , retcd )
```

Argument(s)

i

Default INTEGER scalar. The Fortran record length that is read immediately before.

retcd

Default INTEGER scalar. Variable 0 is returned always.

Example

```
use service_routines,only:redlen
integer :: i,retcd
character(len=10) :: ch
open (10,file='fort.txt')
read (10,*) ch
call redlen(i, retcd)
write (6,*) i
close (10)
end
```

2.394 RENAME Service Function

Description

Renames a file.

Syntax

```
iy = RENAME ( old , new )
```

Argument(s)

old

Default CHARACTER scalar. Path of an existing file.

Its length shall be less than or equal to MAXPATHLEN as defined in the system include.

new

Default CHARACTER scalar. The new path.

Its length shall be less than or equal to MAXPATHLEN as defined in the system include.

Result

Default INTEGER scalar. Zero if successful; otherwise, a system error code.

Remarks

If the file specified exists, RENAME deletes it first.

Example

```
use service_routines,only:rename
iy = rename('xx','yy')
end
```

2.395 REPEAT Intrinsic Function

Description

Concatenate copies of a string.

Class

Transformational function.

Syntax

result = REPEAT (*STRING* , *NCOPIES*)

Argument(s)

STRING

STRING shall be scalar and of type CHARACTER.

NCOPIES

NCOPIES shall be a scalar non-negative INTEGER.

Result

The result is a scalar of type CHARACTER with length equal to *NCOPIES* times the length of *STRING*. Its value is equal to the concatenation of *NCOPIES* copies of *STRING*.

Example

```
character (len=6) :: n
n = repeat('ho',3) ! n is assigned the value 'hohoho'
```

2.396 RESHAPE Intrinsic Function

Description

Construct an array of a specified shape from a given array.

Class

Transformational function.

Syntax

result = RESHAPE (*SOURCE* , *SHAPE* [, *PAD* , *ORDER*])

Required Argument(s)

SOURCE

SOURCE can be of any type. It cannot be scalar. If *PAD* is absent or of size zero, the size of *SOURCE* shall be greater than or equal to the product of the values of the elements of *SHAPE*.

SHAPE

SHAPE shall be an INTEGER array of rank one and of constant size. The size must be a positive number from 1 to 7. It shall not have any negative elements.

Optional Argument(s)

PAD

PAD shall be of the same type and type parameters as *SOURCE*. It cannot be scalar.

ORDER

ORDER shall be of type INTEGER and of the same shape as *SHAPE*. Its value shall be a permutation of (1, 2, ..., *n*) , where *n* is the size of *SHAPE*. If *ORDER* is absent, it is as if it were present with the value (1, 2, ..., *n*).

Result

The result is an array of shape *SHAPE* with the same type and type parameters as *SOURCE*. The elements of the result, taken in permuted subscript order, *ORDER*(1), ..., *ORDER*(*n*) , are those of *SOURCE* in array element order followed if necessary by elements of one or more copies of *PAD* in array element order.

Example

```
integer x(3,2)
x = reshape((/1,2,3,4/), (/3,2/), pad=(/0/))
     ! x is assigned |1 4|
     !               |2 0|
     !               |3 0|
```

2.397 RETURN Statement

Description

The RETURN statement completes execution of a function subprogram or subroutine subprogram and transfers control back to the statement following the procedure invocation.

Syntax

```
RETURN [ scalar-int-expr ]
```

Where:

scalar-int-expr is a scalar INTEGER expression. *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

Remarks

If *scalar-int-expr* is present and has a value n between 1 and the number of asterisks in the subprogram's dummy argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

Example

```
subroutine zee (a, b)
  implicit none
  real, intent(inout) :: a, b
  ...
  if (a>b) then
    return            ! subroutine completed
  else
    a=a*b
    return            ! subroutine completed
  end if
end subroutine zee
```

2.398 REWIND Statement

Description

The REWIND statement positions the specified file at its initial point.

Execution of the REWIND statement performs a wait operation for all asynchronous data transfer operations at the specified device.

Syntax

```
REWIND external-file-unit          or
REWIND ( position-spec-list )
```

Where:

external-file-unit is a scalar INTEGER expression corresponding to the input/output unit number of an external file.

position-spec-list is a comma-separated list of

```
[ UNIT = ] external-file-unit      or
IOMSG = iomsg                      or
IOSTAT = io-stat                   or
ERR = err-label
```

position-spec-list shall contain exactly one *external-file-unit* and may contain at most one of each of the other specifiers.

If the optional characters UNIT= are omitted from the *external-file-unit* specifier, the *external-file-unit* specifier shall be the first item in the *position-spec-list*.

io-stat is a variable of type INTEGER that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, and zero otherwise.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the REWIND statement. If an error condition occurs during execution of the REWIND statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

iomsg must be a scalar default CHARACTER variable. If the error condition occurs during input/output statement execution, an explanatory message is assigned to *iomsg*.

If an error condition does not occur, the *iomsg* value does not change.

A file connected as a direct access cannot be referenced by the REWIND statement.

Remarks

Rewinding a file that is connected but does not exist has no effect.

Example

```
rewind 10     ! file connected to unit 10 rewound
rewind (10, err = 99)
              ! file connected to unit 10 rewound
              ! on error go to label 99
```

2.399 RINDEX Service Function

Description

Locates the index of the last occurrence of a *substring* within a *string*.

Syntax

```
iy = RINDEX ( string , substring )
```

Argument(s)

string

Default CHARACTER scalar. String to search.

substring

Default CHARACTER scalar. Substring to search for.

Result

Default INTEGER scalar. Starting position of the final occurrence of *substring* in *string*. Returns 0 if *substring* does not occur in *string* or *string* is shorter than *substring*.

Example

```
use service_routines,only:rindex
character(len=50) string
character(len=7) :: substr
integer(4) :: offset
string = "Fortran 95 Fortran 90 Fortran77 string scan"
substr = "Fortran"
offset = rindex(string,substr)     ! OFFSET is 23
end
```

2.400 RRSPACING Intrinsic Function

Description

Reciprocal of relative spacing near a given number; *X* divided by SPACING(*X*).

Class

Elemental function.

Syntax

```
result = RRSPACING ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is the reciprocal of the spacing, near *X*, of REAL numbers of the kind of *X*. When there are two of these types of value, the value with the larger absolute value is used. If *X* is IEEE infinity, the result value is zero. If *X* is IEEE NaN, the result value is NaN.

Example

```
r = rrspacing(-4.7)
```

2.401 RSHIFT Intrinsic Function

Description

Arithmetic right shift.

Class

Elemental function.

Syntax

```
result = RSHIFT ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. Its value shall be 0 or positive and less than to the number of bits in *I*.

Result

The result is of type INTEGER and of the same kind as *I*. Its value is the value of *I* arithmetic right shifted by *SHIFT* positions.

Example

```
i = rshift(-7,2)    ! i is assigned the value -2
```

2.402 RTC Service Function

Description

Returns the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

Syntax

```
result = RTC( )
```

Result

REAL(8), scalar. Zero if error; otherwise, number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

Example

```
use service_routines,only:rtc
real(8) :: time1,time2
time1 = rtc()
call sub()
time2 = rtc()
print *,'time spent = ', time2-time1
end
```

2.403 SAME_TYPE_AS Intrinsic Function

Description

The SAME_TYPE_AS intrinsic function queries whether or not the first argument dynamic type is the same as the second argument dynamic type.

Class

Inquiry function.

Syntax

```
result = SAME_TYPE_AS ( A , B )
```

Argument(s)

A

Must be an object that can be extended. If it is a pointer, the associate status cannot be undefined.

B

Must be an object that can be extended. If it is a pointer, the associate status cannot be undefined.

Result

The result is of type default LOGICAL scalar.

True is returned if the *A* dynamic type is the same as the *B* dynamic type.

The dynamic type of a disassociated pointer and the unallocated allocatable type are the declared types for the object.

Example

```
type base
  integer :: base01
end type
type,extends(base):: ext
  integer :: ext01
end type
type (ext),target :: t01a, t01b
class(base),pointer:: p01,p02
p01=>t01a
p02=>t01b
print *,same_type_as(p01, p02)
end
```

2.404 SAVE Statement

Description

The SAVE statement specifies that all objects in the statement retain their association, allocation, definition, and value after execution of a RETURN or END statement of a subprogram.

Syntax

SAVE [[::] *saved-entity-list*]

Where:

saved-entity-list is a comma-separated list of

```
object-name               or
procedure-pointer-name    or
/ common-block-name /
```

object-name is the name of a data object. *object-name* shall not be an object that is in a common block, a dummy argument, a procedure, a function result, an automatic object, or a named constant.

common-block-name is the name of a common block.

Remarks

A SAVE statement without a *saved-entity-list* specifies that all allowable objects in the scoping unit have the SAVE attribute. If a SAVE statement without a *saved-entity-list* is occurs in a scoping unit, no other explicit occurrence of the SAVE attribute or SAVE statement is permitted in the same scoping unit.

Objects declared with the SAVE attribute in a subprogram are shared by all instances of the subprogram.

If a common block is specified in a SAVE statement other than in the main program, it shall be specified in every scoping unit in which it appears except in the main program.

Example

```
subroutine sub( )
save i,j,/myblock/,k  ! i,j,k and common block
                      ! myblock have the save
                      ! attribute
common /myblock/ x1,x2
```

2.405 SCALE Intrinsic Function

Description

Multiply a number by a power of two.

Class

Elemental function.

Syntax

```
result = SCALE ( X , I )
```

Argument(s)

X

X shall be of type REAL.

I

I shall be of type INTEGER.

Result

The result is of the same type and kind as *X*. Its value is $X * 2^I$.

Example

```
x = scale(1.5,3) ! x is assigned the value 12.0
```

2.406 SCAN Intrinsic Function

Description

Scan a string for any one of a set of characters.

Class

Elemental function.

Syntax

```
result = SCAN (STRING,SET[,BACK,KIND])
```

Required Argument(s)

STRING

STRING shall be of type CHARACTER.

SET

SET shall be of the same kind and type as *STRING*.

Optional Argument(s)

BACK

BACK shall be of type LOGICAL.

KIND

KIND shall be of type INTEGER initialization expression.

Result

The result is of type INTEGER. If *KIND* is specified, the kind type parameter of the function result is following to *KIND*. If *KIND* is omitted, the kind type parameter of the function result is default INTEGER.

If *BACK* is absent, or if it is present with the value false, the value of the result is the position number of the leftmost character in *STRING* that is in *SET*. If *BACK* is present with the value true, the value of the result is the position number of the rightmost character in *STRING* that is in *SET*.

Example

```
i = scan ("Lalalalala","la") ! i is assigned the value 2
i = scan ("LalalaLALA","la",back=.true.)
                             ! i is assigned the value 6
```

2.407 SECNDS Service Function

Description

Returns the number of seconds that have elapsed since midnight, less the value of its argument.

Syntax

```
y = SECNDS ( sec )
```

Argument(s)

sec

Default REAL scalar. Number of seconds, precise to a hundredth of a second (0.01), to be subtracted.

Result

Default REAL scalar. Number of seconds that have elapsed since midnight, minus sec, with a precision of a hundredth of a second (0.01).

Example

```
use service_routines,only:secnds
real :: zero=0.0,sec0,sec1
sec0 = secnds(zero)
call sub1()
sec1 = secnds(sec0)
write(6,*) sec1        ! Number of seconds whose SUB1 used
end
```

2.408 SECOND Service Function

Description

Returns the user CPU time since the start of program execution.

Syntax

```
y = SECOND ( )
```

Result

Default REAL scalar. A user time in seconds if successful; otherwise, -1.0.

Example

```
use service_routines,only:second
real :: y1,y2
y1 = second()
do i=1,5000
  write(*,*) i,i*i
end do
y2 = second()
print *,'user time=',y2-y1
end
```

2.409 SELECT CASE Statement

Description

The SELECT CASE statement begins a CASE construct. See "2.56 CASE Construct" for CASE construct.

Syntax

```
[ case-construct-name : ] SELECT CASE ( case-expr )
```

Where:

case-construct-name is an optional name for the CASE construct.

case-expr is a scalar expression of type INTEGER, LOGICAL, or CHARACTER.

2.410 SELECTED_CHAR_KIND Intrinsic Function

Description

The SELECTED_CHAR_KIND intrinsic function returns the kind type parameter value of the character set named by the argument.

Class

Transformational function.

Syntax

```
result = SELECTED_CHAR_KIND ( NAME )
```

Argument(s)

NAME

NAME shall be of type default CHARACTER scalar.

Result

If the value of NAME is 'DEFAULT', the result value is the kind type parameter value of the default CHARACTER type. If the value of NAME is 'ASCII', the result value is the kind type parameter value of the default CHARACTER type. If the value of NAME is 'ISO_10646', the result value is the kind type parameter value of that CHARACTER type. In other cases, the result value is -1. The character string specified at NAME is not case-sensitive, and trailing blanks are skipped.

Example

```
character (kind=selected_char_kind("ASCII")) :: c
! c is of type default CHARACTER.
```

2.411 SELECTED_INT_KIND Intrinsic Function

Description

Kind type parameter of an INTEGER data type that represents all integer values n with $-10^R < n < 10^R$.

Class

Transformational function.

Syntax

```
result = SELECTED_INT_KIND ( R )
```

Argument(s)

R

R shall be a scalar INTEGER.

Result

The result is a scalar of type default INTEGER. Its value is equal to the kind type parameter of the INTEGER data type that accommodates all values n with $-10^R < n < 10^R$. If no such kind is available, the result is -1. If more than one kind is available, the return value is the value of the kind type parameter of the kind with the smallest decimal exponent range.

Example

```
integer (kind=selected_int_kind(3)) :: i,j
! i and j are of a data type with a decimal range of
! at least -1000 to 1000
```

2.412 SELECTED_REAL_KIND Intrinsic Function

Description

Kind type parameter of a REAL data type with decimal precision of at least *P* digits and a decimal exponent range of at least *R*.

Class

Transformational function.

Syntax

```
result = SELECTED_REAL_KIND ( [ P , R ] )
```

Optional Argument(s)

P

P shall be a scalar INTEGER.

R

R shall be a scalar INTEGER.

Result

The result is a scalar of type default INTEGER. Its value is equal to the kind type parameter of the REAL data type with decimal precision of at least *P* digits and a decimal exponent range of at least *R*. If no such kind is available the result is -1 if the precision is not available, -2 if the range is not available, and -3 if neither is available. If more than one kind is available, the return value is the value of the kind type parameter of the kind with the smallest decimal precision.

Remarks

At least one argument shall be present.

Example

```
real (kind=selected_real_kind(3,3)) :: a,b
! a and b are of a data type with a decimal range of
! at least -1000 to 1000 and a precision of at least
! 3 decimal digits
```

2.413 SELECT TYPE Construct

Description

The SELECT TYPE construct selects execution of one of the blocks that comprise it.

The selection is made on the basis of the dynamic type of the selector. The name association with selector is performed using the same method as the ASSOCIATE construct.

Syntax

```
[ select-construct-name: ] SELECT TYPE ( [ associate-name => ] selector )
    [ type-guard-stmt
    block ] ...
END SELECT TYPE [ select-construct-name ]
```

Where:

If *selector* is not a named variable, the syntactic entity '*associate-name* =>' must be present.

If *selector* is not a variable or not a variable with a vector subscript, the *associate-name* cannot appear in the variable definition context.

selector within a SELECT TYPE statement must be polymorphic.

Refer to "2.28.1.1 ASSOCIATE Construct Execution" for the *associate-name* attribute rules.

type-guard-stmt is the type guard statement, and has the following format:

```
TYPE IS( type-spec ) [ select-construct-name ]              or
CLASS IS( derived-type-spec ) [ select-construct-name ]     or
CLASS DEFAULT [ select-construct-name ]
```

type-spec is a type specifier that is either an intrinsic type specifier or a derived type specifier.

derived-type-spec is a derived type specifier. See "2.469 Type Declaration Statement" for intrinsic type specifier, and see "1.5.11.8 Derived Type Specifier" for derived type specifier.

type-spec or *derived-type-spec* must specify that each length type parameter is inherited.

type-spec or *derived-type-spec* cannot specify a sequential derived type or a type having the BIND attribute.

If selector is not infinitely polymorphic, *type-spec* or *derived-type-spec* must specify an extended type of the declared type of that *selector*.

Remarks

In one SELECT TYPE construct, the same type and kind type parameter value cannot be specified in more than one type guard statement starting with TYPE IS, and only one type guard statement starting with CLASS IS can be specified.

There must be one or less CLASS DEFAULT in one SELECT TYPE construct.

If *select-construct-name* is specified in the SELECT TYPE statement of the SELECT TYPE construct, the same *select-construct-name* must be specified in the corresponding END SELECT TYPE statement. If *select-construct-name* is not specified in the SELECT TYPE statement of the SELECT TYPE construct, *select-construct-name* cannot be specified in the corresponding END SELECT TYPE statement. If *select-construct-name* is specified in a type guard statement, the same *select-construct-name* must be specified in the corresponding SELECT TYPE statement.

If the SELECT TYPE construct associate name is not specified, the name that comprises *selector* is used as the associate name.

If selector is not a variable, the selected expression is evaluated when the SELECT TYPE construct is executed.

A maximum of one block can be executed by a SELECT TYPE construct. During execution of that block, *associate-name* specifies the object associated with *selector*.

If a type guard statement starts with TYPE IS, the dynamic type and type parameter values of that statement and *selector* are the same. If it starts with CLASS IS, the dynamic type of that statement matches the dynamic type of the *selector* that is an extended type of that type. In this case, the specified kind type parameter value is the same as the type parameter value corresponding to the dynamic type of the *selector*.

Blocks to be executed are selected in the following order:

1. If *selector* matches a type guard statement starting with TYPE IS, the next block after that statement is executed.
2. If 1. does not apply and *selector* matches only one item in a type guard statement starting with CLASS IS, the next block after that statement is executed.
3. If 1. and 2. do not apply and *selector* matches more than one item in a type guard statement starting with CLASS IS, and if one of those items is an extended type of all the other specified types, the next block after the statement corresponding to that item is executed.
4. If none of the above applies and there is a type guard statement starting with CLASS DEFAULT, the next statement after that statement is executed.

The associated entity within the next block after a type guard statement starting with TYPE IS is not polymorphic, has the type named in the type guard statement, and has the same type parameter value as *selector*.

The associated entity within the next block after a type guard statement starting with CLASS IS is polymorphic and has the declared type named in the type guard statement. The type parameter value of the corresponding *selector* is used as the type parameter value.

The associated entity within the next block after a type guard statement starting with CLASS DEFAULT is polymorphic and has the same declared type as *selector*. The type parameter value of the declared type of the corresponding *selector* is used as the type parameter value.

A type guard statement cannot be used as a jump destination statement. A jump over to an END SELECT TYPE statement can only be from within that SELECT TYPE construct.

Example

```
type p1
  real :: x,y
end type p1
type, extends(p1) :: p2_a
  real :: z
end type p2_a
type, extends(p1) :: p2_b
  integer :: c
end type p2_b
type(p1), target :: t_p1
type(p2_a), target :: t_p2_a
type(p2_b), target :: t_p2_b
class(p1), pointer :: p_p1
p_p1 => t_p2_b
select type (p_p1 )       ! Dynamic type for p_p1 is p2_b
class is ( p1 )           ! p2_b is an extended type of p1
  print *, p_p1%x, p_p1%y ! This block will be executed.
type is ( p2_a )
  print *, p_p1%x, p_p1%y, p_p1%z
end select
end
```

2.414 SEQUENCE Statement

Description

The SEQUENCE statement can only appear in a derived type definition. It specifies that the order of the component definitions is the storage sequence for objects of that type. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
SEQUENCE
```

Remarks

If a derived type definition contains a SEQUENCE statement, the derived type is a sequence type.

If SEQUENCE is present in a derived type definition, all derived types specified in component definitions shall be sequence types.

Example

```
type zee
  sequence       ! zee is a sequence type
  real :: a,b,c  ! a,b,c is the storage sequence for zee
end type zee
```

2.415 SERVICE_ROUTINES Nonstandard Intrinsic Module

Description

This nonstandard intrinsic module enables the reference to the explicit interface of service routines (see "1.12.8 Service Routines").

2.416 SETBIT Service Subroutine

Description

Sets the first argument in word to 1 if state is nonzero and clears it otherwise.

Syntax

```
CALL  SETBIT ( pos , i , status )
```

Argument(s)

pos

Default INTEGER scalar. Bit number to set.

It shall be nonnegative and less than or equal to BIT_SIZE(*i*).

i

Default INTEGER scalar. Variable whose bit is to be set.

status

Default INTEGER scalar. Value to set.

```
=0       :  Sets i in word to 0.
=other   :  Sets i in word to 1.
```

Example

```
use service_routines,only:setbit
integer :: i
i = 10
call setbit(3,i,0)       ! Sets i to 2.
write(6,fmt="(1x,i4)") i
i = 10
call setbit(0,i,1)       ! Sets i to 11.
write(6,fmt="(1x,i4)") i
end
```

2.417 SETRCD Service Subroutine

Description

When the Fortran program is terminated, the lower one-byte value of the argument is set to the return code.

Syntax

```
CALL  SETRCD ( i )
```

Argument(s)

i

Default INTEGER scalar. Return code.

The value is 0 to 255.

Example

```
use service_routines,only:setrcd
read(*,*) ia
if (ia .eq. 99) then
  call setrcd(ia)
else
  print *,'read OK'
end if
end
```

2.418 SET_EXPONENT Intrinsic Function

Description

Model representation of a number with exponent part set to a power of two.

Class

Elemental function.

Syntax

```
result = SET_EXPONENT ( X , I )
```

Argument(s)

X

X shall be of type REAL.

I

I shall be of type INTEGER.

Result

The result is of the same type and kind as *X*. Its value is the $X * \mathrm{RADIX}(X)^{(I-\mathrm{EXPONENT}(X))}$.

Example

```
a = set_exponent (4.6, 2)
```

2.419 SH Service Function

Description

Sends a command to the Bourne shell as if it had been typed at the command line.

Syntax

```
iy = SH ( command )
```

Argument(s)

command

Default CHARACTER scalar. Command to the Bourne shell.

Result

Default INTEGER scalar. Exit status of the shell command.

Remarks

The calling process waits until the command terminates.

Example

```
use service_routines,only:sh
if ( sh('x.sh') /= 0 ) stop 'shell error'
call sub()
end
```

2.420 SHAPE Intrinsic Function

Description

Shape of an array or scalar.

Class

Inquiry function.

Syntax

```
result = SHAPE ( SOURCE [ , KIND ] )
```

Required Argument(s)

SOURCE

SOURCE can be of any type and can be array-valued or scalar. It shall not be an assumed-size array. It shall not be a pointer that is disassociated or an allocatable object that is not allocated.

Optional Argument(s)

KIND

KIND shall be of type INTEGER initialization expression.

Result

The result is of type INTEGER. If *KIND* is specified, the kind type parameter of the function result is following to *KIND*. If *KIND* is omitted, the kind type parameter of the function result is default INTEGER.

The result is an array of rank one whose size is the rank of *SOURCE* and whose value is the shape of *SOURCE*.

Example

```
integer i(3)
integer b(10,-2:10,10)
i = shape(b(1:9,-2:3,:))! i is assigned the value
                        ! (/9,6,10/)
```

2.421 SHIFTA Intrinsic Function

Description

Arithmetic shift to right.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SHIFTA	---	2	INTEGER , INTEGER	INTEGER

```
result = SHIFTA ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. The value shall be less than or equal to BIT_SIZE(*I*), and shall be nonnegative.

Result

The type of the result is the same as the type INTEGER of the *I*.

Remarks

The result is the value of *I* arithmetic shifted by *SHIFT* positions to right.

The type of the result is the same as the type INTEGER of the *I*.

Example

```
k = shifta(-14142, 5)    ! k is assigned the value -442
```

2.422 SHIFTL Intrinsic Function

Description

Left shift.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SHIFTL	---	2	INTEGER , INTEGER	INTEGER

```
result = SHIFTL ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. The value shall be less than or equal to BIT_SIZE(*I*), and shall be nonnegative.

Result

The type of the result is the same as the type INTEGER of the *I*.

Remarks

The result is the value of *I* shifted by *SHIFT* positions to left.

The type of the result is the same as the type INTEGER of the *I*.

Example

```
k = shiftl(223606, 5)    ! k is assigned the value 7155392
```

2.423 SHIFTR Intrinsic Function

Description

Right shift.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SHIFTR	---	2	INTEGER , INTEGER	INTEGER

```
result = SHIFTR ( I , SHIFT )
```

Argument(s)

I

I shall be of type INTEGER.

SHIFT

SHIFT shall be of type INTEGER. The value shall be less than or equal to BIT_SIZE(*I*), and shall be nonnegative.

Result

The type of the result is the same as the type INTEGER of the *I*.

Remarks

The result is the value of *I* shifted by *SHIFT* positions to right.

The type of the result is the same as the type INTEGER of the *I*.

Example

```
k = shiftr(-14142, 5)    ! k is assigned the value 134217286
```

2.424 SHORT Service Function

Description

Converts a value of type default INTEGER to two-byte INTEGER.

Syntax

```
iy = SHORT ( ix )
```

Argument(s)

ix

Default INTEGER scalar. Value to be converted.

Result

Two-byte INTEGER scalar. Equal to the lower 16 bits of *ix*.

Example

```
use service_routines,only:short
integer(kind=2) :: ix
integer :: ix4
ix = short(ix4)
end
```

2.425 SIGN Intrinsic Function

Description

Transfer of sign.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SIGN	---	2	One-byte INTEGER , One-byte INTEGER	One-byte INTEGER
	I2SIGN		Two-byte INTEGER , Two-byte INTEGER	Two-byte INTEGER
	IISIGN		Two-byte INTEGER , Two-byte INTEGER	Two-byte INTEGER
	ISIGN		Four-byte INTEGER , Four-byte INTEGER	Four-byte INTEGER
	JISIGN		Four-byte INTEGER , Four-byte INTEGER	Four-byte INTEGER
	---		Eight-byte INTEGER , Eight-byte INTEGER	Eight-byte INTEGER

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	SIGN		Single-prec REAL , Single-prec REAL	Single-prec REAL
	DSIGN		Double-prec REAL , Double-prec REAL	Double-prec REAL
	QSIGN		Quad-prec REAL , Quad-prec REAL	Quad-prec REAL

```
result = SIGN ( A , B )
```

Argument(s)

A

A shall be of type INTEGER or REAL.

B

B shall be of the same type and kind as *A*.

Result

The result is of the same type and kind as *A*. Its value is the |*A*|, if *B* is greater than or equal to zero; and -|*A*|, if *B* is less than zero.

Remarks

The generic name, SIGN, may be used with any real argument.

The type of the result of each function is the same as the type of the arguments.

Example

```
i = sign (30,-2) ! i is assigned the value -30
```

2.426 SIGNAL Service Function

Description

Issues a signal. The arguments of signal are *i*, *func*, and *flag*.

Syntax

```
iy = SIGNAL ( i , func , flag )
```

Argument(s)

i

Default INTEGER scalar. Signal number.

func

The name of the signal handling routine.

flag

Default INTEGER scalar. The value is passed to the system as the signal action definition.
The value is as follows:

```
<0  :  Uses func as a signal handling routine.
=0  :  Uses the default action.
=1  :  Ignores a signal.
>1  :  func is ignored and the value is passed to the system.
```

Result

Default INTEGER scalar. Return the value of *func* or *flag* before signal.

Return the before *flag* value if it is 0 or 1.

Return the before *func* value if before *flag* value is negative.

Return system error code if *i* is ineffective or *flag* greater than 1.

Example

```
use service_routines,only:signal
integer(kind=2),external :: func
integer :: iy
iy = signal(8,func,-1)
end
```

2.427 SIN Intrinsic Function

Description

Sine of an argument in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SIN	---	1	REAL or COMPLEX	REAL or COMPLEX
	SIN		Single-prec REAL	Single-prec REAL
	DSIN		Double-prec REAL	Double-prec REAL
	QSIN		Quad-prec REAL	Quad-prec REAL
	CSIN		Single-prec COMPLEX	Single-prec COMPLEX
	CDSIN		Double-prec COMPLEX	Double-prec COMPLEX
	CQSIN		Quad-prec COMPLEX	Quad-prec COMPLEX

result = SIN (*X*)

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the sine of *X*.

Remarks

SIN, DSIN, QSIN, CSIN, CDSIN, and CQSIN evaluate the sine of a REAL or COMPLEX data in radians.

For a single precision REAL argument, the domain shall be ABS(*X*) < 8.23E+05 .

For a double precision REAL argument, the domain shall be DABS(*X*) < 3.53D+15 .

For a quadruple precision REAL argument, the domain shall be QABS(*X*) < $2.0Q0^{62} * \pi$.

For a single precision COMPLEX argument, the domain shall be ABS(REAL(*X*)) < 8.23E+05 and ABS(AIMAG(*X*)) < 89.415E0 .

For a double precision COMPLEX argument, the domain shall be DABS(DREAL(*X*)) < 3.53D+15 and DABS(DIMAG(*X*)) < 710.475D0 .

For a quadruple precision COMPLEX argument, the domain shall be QABS(QREAL(*X*)) < $2.0Q0^{62} * \pi$ and QABS(QIMAG(*X*)) < 11357.125Q0 .

The generic name, SIN, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = sin(.5)
```

2.428 SIND Intrinsic Function

Description

Sine of an argument in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SIND	---	1	REAL	REAL
	SIND		Single-prec REAL	Single-prec REAL
	DSIND		Double-prec REAL	Double-prec REAL
	QSIND		Quad-prec REAL	Quad-prec REAL

```
result = SIND ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the sine of *X*.

Remarks

SIND, DSIND, and QSIND evaluate the sine of a REAL data in degrees. The result has the value SIN(π /180**X*).

For a single precision REAL argument, the domain shall be ABS(*X*) < 4.72E+07 .

For a double precision REAL argument, the domain shall be DABS(*X*) < 2.03D+17.

For a quadruple precision REAL argument, the domain shall be QABS(*X*) < 2.0Q0^{62}*180.

The generic name, SIND, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = sind(.5)
```

2.429 SINH Intrinsic Function

Description

Hyperbolic sine.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SINH	---	1	REAL or COMPLEX	REAL or COMPLEX
	SINH		Single-prec REAL	Single-prec REAL

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
	DSINH		Double-prec REAL	Double-prec REAL
	QSINH		Quad-prec REAL	Quad-prec REAL

```
result = SINH ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the hyperbolic sine of *X*.

Remarks

SINH evaluates the hyperbolic sine of REAL or COMPLEX data. DSINH and QSINH evaluate the hyperbolic sine of a REAL data.

For a single precision REAL argument, the domain shall be ABS(*X*) < 89.415E0 .

For a double precision REAL argument, the domain shall be DABS(*X*) < 710.475D0.

For a quadruple precision REAL argument, the domain shall be QABS(*X*) < 11357.125Q0.

For a single precision COMPLEX argument, the domain shall be ABS(REAL(*X*))<89.415E0 and ABS(AIMAG(*X*))<8.23E+05 .

For a double precision COMPLEX argument, the domain shall be DABS(DREAL(*X*)) < 710.475D0 and DABS(DIMAG(*X*)) < 3.53D+15 .

For a quadruple precision COMPLEX argument, the domain shall be QABS(QREAL(*X*)) < 11357.125Q0 and QABS(QIMAG(*X*)) < $2.0Q0^{62} * \pi$.

The generic name, SINH, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = sinh(.5)
```

2.430 SINQ Intrinsic Function

Description

Sine of an argument in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SINQ	---	1	REAL or COMPLEX	REAL or COMPLEX
	SINQ		Single-prec REAL	Single-prec REAL
	DSINQ		Double-prec REAL	Double-prec REAL
	QSINQ		Quad-prec REAL	Quad-prec REAL
	CSINQ		Single-prec COMPLEX	Single-prec COMPLEX
	CDSINQ		Double-prec COMPLEX	Double-prec COMPLEX
	CQSINQ		Quad-prec COMPLEX	Quad-prec COMPLEX

```
result = SINQ ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the sine of *X*.

Remarks

SINQ, DSINQ, QSINQ, CSINQ, CDSINQ, and CQSINQ evaluate the sine of a REAL or COMPLEX data in quadrants. The result has the value SIN(π /2***X*).

For a single precision COMPLEX argument, the domain shall be ABS(REAL(*X*)) < 5.24E+05 and ABS(AIMAG(*X*)) < 56.92E0 .

For a double precision COMPLEX argument, the domain shall be DABS(DREAL(*X*)) < 2.25D+15 and DABS(DIMAG(*X*)) < 452.305D0 .

For a quadruple precision COMPLEX argument, the domain shall be QABS(QREAL(*X*)) < $2.0Q0^{63}$ and QABS(QIMAG(*X*)) < 7230.125Q0 .

The generic name, SINQ, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = sinq(.5)
```

2.431 SIZE Intrinsic Function

Description

Size of an array or a dimension of an array.

Class

Inquiry function.

Syntax

result = SIZE (*ARRAY* [, *DIM* , *KIND*])

Required Argument(s)

ARRAY

ARRAY can be of any type. It shall not be a scalar and shall not be a pointer that is disassociated or an allocatable object that is not allocated.

Optional Argument(s)

DIM

DIM shall be of type INTEGER and shall be a dimension of *ARRAY*. If *ARRAY* is assumed-size, *DIM* shall be present and less than the rank of *ARRAY*.

KIND

KIND shall be of type INTEGER initialization expression.

Result

The result is a scalar of type INTEGER. If *KIND* is specified, the kind type parameter of the function result is following to *KIND*. If *KIND* is omitted, the kind type parameter of the function result is default INTEGER.

If *DIM* is present, the result is the extent of dimension *DIM* of *ARRAY*. If *DIM* is absent, the result is the number of elements in *ARRAY*.

Example

```
integer, dimension (3,-4:0) :: i
integer :: k,j
j = size (i)     ! j is assigned the value 15
k = size (i, 2)  ! k is assigned the value 5
```

2.432 SIZEOF Intrinsic Function

Description

The number of bytes of the argument.

Class

Inquiry function.

Syntax

result = SIZEOF (*X*)

Argument(s)

X

X can be of any type. It shall not be an assumed-size array, a pointer that is disassociated or an allocatable object that is not allocated.

Result

Eight-byte INTEGER. It is a scalar. If the argument is scalar, size of the object is returned. If the argument is array, a result of the multiplication of the size of one element and the number of elements is returned.

Example

```
integer, dimension(3,-4:0) :: i
integer :: k
k = sizeof (i)    ! k is assigned the value 60.
```

2.433 SLEEP Service Subroutine

Description

Suspends the execution of a process for a specified interval.

Syntax

CALL SLEEP (*i*)

Argument(s)

i

Default INTEGER scalar. Length of time, in seconds, to suspend the calling process.

Remarks

The actual time can be up to 1 second less than *i* due to granularity of the system timekeeping.

Example

```
use service_routines,only:sleep
call sleep(200)
end
```

2.434 SLITE Service Subroutine

Description

SLITE turns sense lights on or off.

Syntax

```
CALL SLITE ( i )
```

Argument(s)

i

Default INTEGER scalar. Specifies sense lights. It is no effect to specify other value.

```
=0  :  Turns off the four sense lights.
=1  :  Turns on sense light 1.
=2  :  Turns on sense light 2.
=3  :  Turns on sense light 3.
=4  :  Turns on sense light 4.
```

Example

```
use service_routines,only:slite
call slite(1)
end
```

2.435 SLITET Service Subroutine

Description

SLITET tests if the sense light specified in the first argument is on or off. If the specified sense light is on, the value 1 is assigned to the second argument. If the specified sense light is off, the value 2 is assigned to the second argument. Then, the corresponding sense light is turned off.

Syntax

```
CALL SLITET ( i , j )
```

Argument(s)

i

Default INTEGER scalar. Specifies sense lights. It is no effect to specify other value.

```
=1  :  Tests the sense light 1.
=2  :  Tests the sense light 2.
=3  :  Tests the sense light 3.
=4  :  Tests the sense light 4.
```

j

Default INTEGER scalar. Returns a sense light status. It is undefined when invalid value is specified in *i*.

```
=1  :  Sense light was being light on.
=2  :  Sense light was being light off.
```

Example

```
use service_routines,only:slitet
integer :: j
call slitet(1,j)
print *,j
end
```

2.436 SPACING Intrinsic Function

Description

Absolute spacing near a given number; the difference between *X* and the next representable number whose absolute value is greater than that of *X*.

Class

Elemental function.

Syntax

```
result = SPACING ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is the spacing of REAL values, of the kind of *X*, near *X*.

If *X* is IEEE infinity, the result value is positive infinity. If *X* is IEEE NaN, the result value is NaN.

Example

```
x = spacing(4.7)
```

2.437 SPREAD Intrinsic Function

Description

Adds a dimension to an array by adding copies of a data object along a given dimension.

Class

Transformational function.

Syntax

```
result = SPREAD ( SOURCE , DIM , NCOPIES )
```

Argument(s)

SOURCE

SOURCE can be of any type and can be scalar or array-valued. Its rank shall be less than seven.

DIM

DIM shall be a scalar of type INTEGER with a value in the range 1 <= *DIM* <= *n*+1 , where *n* is the rank of *SOURCE*.

NCOPIES

NCOPIES shall be a scalar of type INTEGER.

Result

The result is an array of the same type and kind as *SOURCE* and of rank *n* + 1, where *n* is the rank of *SOURCE*. If *SOURCE* is scalar, the shape of the result is MAX(*NCOPIES* , 0) and each element of the result has a value equal to *SOURCE*. If *SOURCE* is array-valued with shape (d_1, d_2, ..., d_n) , the shape of the result is (d_1, d_2, ..., d_{DIM-1}, MAX(*NCOPIES*,0), d_{DIM+1}, ..., d_n) and the element of the result with subscripts (r_1, r_2, ..., r_{n+1}) has the value *SOURCE*(r_1, r_2, ..., r_{DIM-1}, r_{DIM+1}, ..., r_{n+1}).

Example

```
integer, dimension(2) :: b=(/1,2/)
integer, dimension(2,3) :: a
a = spread(b,2,3) ! a is assigned |1 1 1|
                  !               |2 2 2|
```

2.438 SQRT Intrinsic Function

Description

Square Root.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
SQRT	---	1	REAL or COMPLEX	REAL or COMPLEX
	SQRT		Single-prec REAL	Single-prec REAL
	DSQRT		Double-prec REAL	Double-prec REAL
	QSQRT		Quad-prec REAL	Quad-prec REAL
	CSQRT		Single-prec COMPLEX	Single-prec COMPLEX
	CDSQRT		Double-prec COMPLEX	Double-prec COMPLEX
	CQSQRT		Quad-prec COMPLEX	Quad-prec COMPLEX

```
result = SQRT ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX. If *X* is REAL, its value shall be greater than or equal to zero.

Result

The result is of the same kind and type as *X*. If *X* is of type REAL, the result value is a REAL representation of the square root of *X*. If *X* is of type COMPLEX, the result value is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Remarks

SQRT, DSQRT, QSQRT, CSQRT, CDSQRT, and CQSQRT evaluate the square root of REAL or COMPLEX data. If *X* is of type REAL, its value shall be *X*>=0.0 .

The generic name, SQRT, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
x = sqrt(16.0) ! x is assigned the value 4.0
```

2.439 STAT Service Function

Description

Returns information a file.

Syntax

```
iy = STAT ( name , status )
```

Argument(s)

name

Default CHARACTER scalar. A file name.

status

Default INTEGER and rank one. Its size shall be at least 13. The values returned in *status* are as follows:

```
status(1)  : File mode.
status(2)  : Inode number.
status(3)  : ID of device containing a directory entry for this file.
status(4)  : ID of device. This entry is defined only for char special or block special files.
```

```
status(5)  : Number of links.
status(6)  : User ID of the file's owner.
status(7)  : Group ID of the file's group.
status(8)  : File size in bytes.
status(9)  : Time of last access.
status(10) : Time of last data modification.
status(11) : Time of last file status change.
status(12) : Preferred I/O block size.
status(13) : Number of 512 byte blocks allocated.
```

Result

Default INTEGER scalar. Zero if successful; otherwise, a system error code.

Example

```
use service_routines,only:stat
integer :: st(13)
print *,stat('xxx.dat',st)
end
```

2.440 STAT64 Service Function

Description

Returns information a file.

Syntax

```
iy = STAT64 ( name , status )
```

Argument(s)

name

Default CHARACTER scalar. A file name.

status

Eight-byte INTEGER and rank one. Its size shall be at least 13. The values returned in *status* are as follows:

```
status(1)  : File mode.
status(2)  : Inode number.
status(3)  : ID of device containing a directory entry for this file.
status(4)  : ID of device. This entry is defined only for char special or block special files.
status(5)  : Number of links.
status(6)  : User ID of the file's owner.
status(7)  : Group ID of the file's group.
status(8)  : File size in bytes.
status(9)  : Time of last access.
status(10) : Time of last data modification.
status(11) : Time of last file status change.
status(12) : Preferred I/O block size.
status(13) : Number of 512 byte blocks allocated.
```

Result

Default INTEGER scalar. Zero if successful; otherwise, a system error code.

Example

```
use service_routines,only:stat64
integer(kind=8) :: st(13)
print *,stat64('xxx.dat',st)
end
```

2.441 Statement Function Statement (obsolescent feature)

Description

A statement function is a function defined by a single statement.

Syntax

```
function-name ( [ dummy-arg-name-list ] ) = scalar-expr
```

Where:

function-name is the name of the statement function being defined.

dummy-arg-name-list is a comma-separated list of dummy argument names.

scalar-expr is a scalar expression.

Remarks

scalar-expr can be composed only of literal or named constants, references to variables, references to functions and function dummy procedures, and intrinsic operators. 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.

Each scalar variable reference in *scalar-expr* shall be either a reference to a dummy argument of the statement function or a reference to a variable local to the same scoping unit as the statement function statement.

The dummy arguments have a scope of the statement function statement.

A statement function shall not be supplied as a procedure argument.

The value of a statement function reference is obtained by evaluating the expression using the values of the actual arguments for the values of the corresponding dummy arguments and, if necessary, converting the result to the declared type and type attributes of the function.

Example

```
mean(i,j) = (i + j) / 2
n = mean(2,4)           ! n is assigned the value 3
```

2.442 STATIC Statement

Description

The STATIC statement declares specified variable to be in static memory and variables have the SAVE attribute.

Syntax

```
STATIC [ [ :: ] object-name-list ]
```

Where:

object-name-list is a comma-separated list of an object name.

object-name shall not be a dummy argument, a procedure, a function result, an automatic object, an equivalence object, or a common block object.

Remarks

If *object-name-list* is not present in the STATIC statement, all the allowable local variables have the SAVE attribute implicitly.

A STATIC statement may not be specified in a module specification part and block data program unit.

Example

```
subroutine sub
  static :: i,j  ! i and j are on static memory
```

2.443 STOP Statement

Description

The STOP statement normally terminates execution of the program.

Syntax

```
STOP [ stop-code ]
```

Where:

stop-code is

```
scalar-char-constant                                  or
digit [ digit [ digit [ digit [ digit ] ] ] ]         or
scalar-default-char-initialization-expr               or
scalar-int-initialization-expr
```

scalar-char-constant is a scalar default CHARACTER constant.

digit is a digit.

scalar-default-char-initialization-expr is a scalar default CHARACTER initialization expression.

scalar-int-initialization-expr is a scalar INTEGER initialization expression.

Remarks

When a STOP statement is reached, the diagnostic message and the optional *stop-code* if present are written to the standard error file and the executing program is normally terminated.

Example

```
if (a>b) then
  stop          ! program execution terminated
end if
```

2.444 STORAGE_SIZE Intrinsic Function

Description

Storage size in bits.

Class

Inquiry function.

Syntax

```
result = STORAGE_SIZE ( A [ , KIND ] )
```

Required Argument(s)

A

A can be of any type. If *A* is polymorphic, it shall not be an undefined pointer. If *A* has any deferred type parameter, it shall not be an unallocated variable or a disassociated or undefined pointer.

Optional Argument(s)

KIND

KIND shall be a scalar INTEGER initialization expression.

Result

The result is scalar of the INTEGER type. If *KIND* is present the result kind is *KIND*; otherwise it is the default INTEGER kind.

Remarks

The result value is the size expressed in bits for dynamic type of *A*. If *A* is an array, the result is size of an element.

Example

```
integer,dimension(100) :: i
k = storage_size(i)    ! k is assigned the value 32
```

2.445 STRUCTURE Statement

Description

The STRUCTURE statement begins a derived type definition. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
STRUCTURE [ / type-name / ] [ component-list ]
```

Where:

type-name is the name of the derived type being defined.

type-name shall not be the name of an intrinsic type nor of another accessible derived type name.

IF derived type definitions using STRUCTURE statements are nested, only the / *type-name* / argument in the inner definition may be omitted.

The *component-list* can be specified only in the inner STRUCTURE statement of the nested definition. The *component* arguments are structure names if a derived type defined by this STRUCTURE statement.

The derived type defined by STRUCTURE statement is a sequence type.

Example

```
structure /complex_element/
  union
    map
      real :: real,imag
    end map
    map
      complex :: complex
    end map
  end union
end structure
record /complex_element/ x
x%real = 2.0
x%imag = 3.0
print *,x%complex     ! complex has the value (2.0,3.0)
```

2.446 SUBROUTINE Statement

Description

The SUBROUTINE statement begins a subroutine subprogram and specifies characteristics of the subroutine. See "1.12.2 Subroutine Subprogram" for subroutine subprogram.

Syntax

```
[ prefix-spec ] ... SUBROUTINE subroutine-name [ ( [ dummy-arg-list ] )  &
&  proc-language-binding-spec ]
```

Where:

prefix-spec is

```
RECURSIVE        or
PURE             or
ELEMENTAL
```

subroutine-name is the name of the subroutine.

dummy-arg-list is a comma-separated list of

```
dummy-arg-name              or
* (obsolescent feature)
```

dummy-arg-name is the name of the dummy argument.

proc-language-binding-spec is

```
BIND ( C [, NAME = scalar-char-initialization-expr ] )
```

Remarks

If the RECURSIVE is present, the subroutine subprogram is a recursive procedure. If the PURE is present, the subroutine subprogram is a pure subprogram. If the ELEMENTAL is present, the subroutine subprogram is an elemental subprogram. See "1.12.3 Recursive Procedures" for recursive procedure, "1.12.4 Pure Procedures" for pure subprogram, "1.12.5 Elemental Procedures" for elemental subprogram.

Example

```
pure subroutine zee (var1, var2)
...
end subroutine
```

2.447 SUM Intrinsic Function

Description

Sum of elements of an array, along a given dimension, for which a mask is true.

Class

Transformational function.

Syntax

```
result = SUM ( ARRAY [ , MASK ] )         or
result = SUM ( ARRAY , DIM [ , MASK ] )
```

Required Argument(s)

ARRAY

ARRAY shall be of type INTEGER, REAL, or COMPLEX. It shall not be scalar.

Optional Argument(s)

DIM

DIM shall be a scalar INTEGER in the range 1 <= *DIM* <= *n*, where *n* is the rank of *ARRAY*. The corresponding dummy argument shall not be an optional dummy argument.

MASK

MASK shall be of type LOGICAL and shall be conformable with *ARRAY*.

Result

The result is of the same type and kind as *ARRAY*. It is scalar if *DIM* is absent or if *ARRAY* has rank one; otherwise the result is an array of rank *n*-1 and of shape $(d_1,d_2,...,d_{DIM-1},d_{DIM+1},...,d_n)$ where $(d_1,d_2,...,d_n)$ is the shape of *ARRAY*. If *DIM* is absent, the value of the result is the sum of the values of all the elements of *ARRAY*. If *DIM* is present, the value of the result is the sum of the values of all elements of *ARRAY* along dimension *DIM*. If *MASK* is present, the elements of *ARRAY* for which *MASK* is false are not considered.

Example

```
integer j(2)
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
! m is the array |1 3|
!                |2 4|
i = sum(m)          ! i is assigned the value 10
```

```
j = sum(m,dim=1)    ! j is assigned the value [3,7]
k = sum(m,mask=m>2) ! k is assigned the value 7
```

2.448 SYMLNK Service Function

Description

Makes a symbolic link to an existing file.

Syntax

```
iy = SYMLNK ( path1 , path2 )
```

Argument(s)

path1

Default CHARACTER scalar. A pathname of an existing file.

path2

Default CHARACTER scalar. A pathname to be linked to file *path1*. *path2* shall not already exist.

Result

Zero if the call was successful; otherwise, a system error code.

Example

```
use service_routines,only:symlnk
i=symlnk('libx.so.1','../libx.so')
end
```

2.449 SYNC ALL Statement

Description

SYNC ALL statement causes synchronization among all images.

Syntax

```
SYNC ALL [ ( [ sync-stat-list ] ) ]
```

Where:

sync-stat-list is a comma separated list of

```
STAT = stat-variable            or
ERRMSG = errmsg-variable
```

No specifier must appear in *sync-stat-list* repeatedly. See "1.18.2 Synchronization Status Specifier" for *stat-variable* and *errmsg-variable*.

Remarks

Execution of SYNC ALL statement causes synchronization among all images. The execution of segments on M that succeed a SYNC ALL statement is delayed until each image completes execution of SYNC ALL statements same times as image M had executed. Segments that precede a SYNC ALL statement precede segments on other images that succeed the SYNC ALL statement.

Example

```
integer :: x[*]
x = 1
sync all
print *,x[1]  ! "1" is output on each image
end
```

2.450 SYNC IMAGES Statement

Description

SYNC IMAGES statement does synchronization between images.

Syntax

```
SYNC IMAGES ( image-set [ , sync-stat-list ] )
```

Where:

image-set is

```
int-expr                    or
*
```

int-expr must be a scalar or of rank one.

If *int-expr* is an array, each element must be positive, less than or equal to the number of images, and must not appear repeatedly.

If *int-expr* is a scalar, the value must be positive and less than or equal to the number of images.

An asterisk in *image-set* means all images.

sync-stat-list is a comma separated list of

```
STAT = stat-variable          or
ERRMSG = errmsg-variable
```

No specifier must appear in *sync-stat-list* repeatedly. See "1.18.2 Synchronization Status Specifier" for *stat-variable* and *errmsg-variable*.

Remarks

Execution of a SYNC IMAGES statement causes synchronization between executing image and the images in image set. SYNC IMAGES statements on image T and image M correspond if the number of execution of SYNC IMAGES statements that include T in its image set on image M and the number of the execution of SYNC IMAGES statements that include M in its image set on image T is same. Segments that precede a SYNC IMAGES statement precede segments that succeed the SYNC IMAGES statement.

Example

```
if (this_image() == 1 .or. this_image() == 2) then
  sync images([1,2])   ! a synchronization between image 1 and image 2 occurs
end if
end
```

2.451 SYNC MEMORY Statement

Description

SYNC MEMORY statement terminates a segment and starts another segment.

Syntax

```
SYNC MEMORY [ ( [ sync-stat-list ] ) ]
```

Where:

sync-stat-list is a comma separated list of

```
STAT = stat-variable
ERRMSG = errmsg-variable
```

No specifier must appear in *sync-stat-list* repeatedly. See "1.18.2 Synchronization Status Specifier" for *stat-variable* and *errmsg-variable*.

Remarks

SYNC MEMORY statement terminates a segment and starts another segment. These two segments can be ordered by user-defined way. See "1.18.1 Segment" for segments.

In the following case, an action to variable X on image Q precedes an action to variable Y on image Q:

- the X on image Q is defined, referenced, or become undefined, or the allocation status, pointer association status, bounds, dynamic type, or type parameter of X is changed or inquired about by a statement on image P,
- the statement precedes a SYNC MEMORY statement, and
- the Y on image Q is defined, referenced, or become undefined, or the allocation status, pointer association status, bounds, dynamic type, or type parameter of X is changed or inquired about by a statement on image P that succeeds the SYNC MEMORY statement.

In the following case, user-defined ordering of segment P_i on image P to precedes segment Q_i on image Q:

- an image control statement is executed that terminates segment P_i, and then statements that starts cooperative synchronization between image P and image Q is executed on image P, and
- the cooperative synchronization is completed on image Q, and then an image control statement that starts segment Q_j is executed on image Q.

The cooperative synchronization between image P and image Q must include the dependency that forces the statement on P that starts the synchronization precedes the statement on image Q that completes the synchronization.

Example

```
integer, save :: i[*], k[*]
i=1
sync memory
k=1            ! "i=1" precedes this statement on each image
end
```

2.452 SYSTEM Service Function

Description

Sends a command to the shell as if it had been typed at the command line.

Syntax

```
iy = SYSTEM ( ch )
```

Argument(s)

ch

Default CHARACTER scalar. Operating system command.

Result

Default INTEGER scalar. Exit status of the shell command.

Remarks

The calling process waits until the command terminates.

Example

```
use service_routines,only:system
if ( system('x.sh') /= 0 ) stop 'shell error'
call sub()
end
```

2.453 SYSTEM_CLOCK Intrinsic Subroutine

Description

INTEGER data from the real-time clock.

Class

Subroutine.

Syntax

```
CALL  SYSTEM_CLOCK ( [ COUNT , COUNT_RATE , COUNT_MAX ] )
```

Optional Argument(s)

COUNT

COUNT shall be a scalar of type default INTEGER. It is an INTENT(OUT) argument. Its value is set to the current value of the processor clock or to -HUGE(0) if no clock is available.

COUNT_RATE

COUNT_RATE shall be a scalar of type default INTEGER or REAL. It is an INTENT(OUT) argument. It is set to the number of processor clock counts per second, or to zero if there is no clock.

COUNT_MAX

COUNT_MAX shall be a scalar of type default INTEGER. It is an INTENT(OUT) argument. It is set to the maximum value that *COUNT* can have, or zero if there is no clock.

Example

```
integer c,cr,cm
call system_clock(c, cr, cm) ! c is set to current
                             ! value of processor
                             ! clock. cr is set to
                             ! the count_rate, and cm
                             ! is set to the
                             ! count_max
```

2.454 TAN Intrinsic Function

Description

Tangent of an argument in radians.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
TAN	---	1	REAL or COMPLEX	REAL or COMPLEX
	TAN		Single-prec REAL	Single-prec REAL
	DTAN		Double-prec REAL	Double-prec REAL
	QTAN		Quad-prec REAL	Quad-prec REAL

```
result = TAN ( X )
```

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the tangent of *X*.

Remarks

TAN evaluates the tangent of REAL or COMPLEX data.

DTAN and QTAN evaluate the tangent of a REAL data in radians.

For a single precision REAL argument, the domain shall be ABS(*X*) < 8.23E+05 .

For a double precision REAL argument, the domain shall be DABS(*X*) < 3.53D+15.

For a quadruple precision REAL argument, the domain shall be QABS(*X*) < $2.0Q0^{62} * \pi$.

For a single precision COMPLEX argument, the domain shall be ABS(REAL(*X*)) < 8.23E+05.

For a double precision COMPLEX argument, the domain shall be DABS(DREAL(*X*)) < 3.53D+15.

For a quadruple precision COMPLEX argument, the domain shall be QABS(QREAL(*X*)) < $2.0Q0^{62} * \pi$.

The generic name, TAN, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = tan(.5)
```

2.455 TAND Intrinsic Function

Description

Tangent of an argument in degrees.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
TAND	---	1	REAL	REAL
	TAND		Single-prec REAL	Single-prec REAL
	DTAND		Double-prec REAL	Double-prec REAL
	QTAND		Quad-prec REAL	Quad-prec REAL

```
result = TAND ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is a REAL representation of the tangent of *X*.

Remarks

TAND, DTAND, and QTAND evaluate the tangent of a REAL data in degrees. The result has the value TAN($\pi/180*X$) .

For a single precision REAL argument, the domain shall be ABS(*X*) < 4.72E+07 .

For a double precision REAL argument, the domain shall be DABS(*X*) < 2.03D+17.

For a quadruple precision REAL argument, the domain shall be QABS(*X*) < $2.0Q0^{62}*180$.

The generic name, TAND, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = tand(.5)
```

2.456 TANH Intrinsic Function

Description

Hyperbolic tangent.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
TANH	---	1	REAL or COMPLEX	REAL or COMPLEX
	TANH		Single-prec REAL	Single-prec REAL
	DTANH		Double-prec REAL	Double-prec REAL
	QTANH		Quad-prec REAL	Quad-prec REAL

result = TANH (*X*)

Argument(s)

X

X shall be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *X*. Its value is a REAL or COMPLEX representation of the hyperbolic tangent of *X*.

Remarks

TANH evaluates the hyperbolic tangent of REAL or COMPLEX data.DTANH and QTANH evaluate the hyperbolic tangent of a REAL data.

For a single precision COMPLEX argument, the domain shall be ABS(AIMAG(*X*)) < 8.23E+05.

For a double precision COMPLEX argument, the domain shall be DABS(DIMAG(*X*)) < 3.53D+15.

For a quadruple precision COMPLEX argument, the domain shall be QABS(QIMAG(*X*)) < $2.0Q0^{62} * \pi$.

The generic name, TANH, may be used with any REAL or COMPLEX argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = tanh(.5)
```

2.457 TANQ Intrinsic Function

Description

Tangent of an argument in quadrants.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
TANQ	---	1	REAL	REAL
	TANQ		Single-prec REAL	Single-prec REAL
	DTANQ		Double-prec REAL	Double-prec REAL
	QTANQ		Quad-prec REAL	Quad-prec REAL

```
result = TANQ ( X )
```

Argument(s)

X

X shall be of type REAL.

Result

The result is of the same type and kind as *X*. Its value is a REAL representation of the tangent of *X*.

Remarks

TANQ, DTANQ, and QTANQ evaluate the tangent of a REAL data in quadrants. The result has the value TAN(π /2*X).

The generic name, TANQ, may be used with any REAL argument.

The type of the result of each function is the same as the type of the argument.

Example

```
r = tanq(.5)
```

2.458 TARGET Statement

Description

The TARGET statement specifies a list of object names that have the target attribute and thus can have pointers associated with them.

Syntax

```
TARGET [ :: ] target-decl-list
```

target-decl is

```
object-name [ ( array-spec ) ] [ left-square-bracket coarray-spec right-square-bracket ]
```

Where:

object-name is the name of a data object.

array-spec is an array specification. See "2.113 DIMENSION Statement" for array specification.

coarray-spec is a coshape coarray specifier. See "1.17.1 Coarray Specifier" for coshape coarray specifier.

left-square-bracket is a left-side square bracket. The left-side square bracket is '['.

right-square-bracket is a right-side square bracket. The right-side square bracket is ']'.

Example

```
target a,b,c  ! a,b, and c have the target attribute
```

2.459 THIS_IMAGE Intrinsic Function

Description

Returns the cosubscript(s) for the invoking image.

Class

Transformational function.

Syntax

```
result = THIS_IMAGE ( )                    or
result = THIS_IMAGE ( COARRAY [, DIM ] )
```

Optional Argument(s)

COARRAY

COARRAY must be a coarray. If it is allocatable, it must be allocated.

DIM

DIM must be a default integer scalar in the range 1 <= *DIM* <= n, where n is the corank of *COARRAY*. The corresponding actual argument must not be an optional dummy argument.

Result

It is default integer. If *COARRAY* is not specified, it is the value of the index of the invoking image. If *COARRAY* is specified and *DIM* is not specified, it is an array of rank one, which specifies the invoking image as cosubscripts for *COARRAY*. If both *COARRAY* and *DIM* are specified, it is a scalar with the value of cosubscript *DIM* in the sequence of cosubscript values for *COARRAY* that specifies the invoking images.

Example

```
if (this_image() == 1) then
  print *,'image 1' ! executed by only image 1
end if
```

2.460 TIME Service Function

Description

Returns the system time, in seconds, since 00:00:00 Greenwich mean time, January 1, 1970.

Syntax

```
iy = TIME ( )
```

Result

Default INTEGER scalar. Number of seconds that have elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

Example

```
use service_routines,only:time
integer :: iy
iy = time()
end
```

2.461 TIMEF Service Function

Description

Returns the number of seconds since the first time it is called, or zero.

Syntax

```
dy = TIMEF ( )
```

Result

Double precision REAL scalar. Number of seconds that have elapsed since the first time TIMEF( ) was called. The first time it is called, TIMEF returns 0.0D0.

Example

```
use service_routines,only:timef
real(kind=8) :: y
y = timef()
do i=1,5000
  write(*,*) i,i*i
end do
y = timef()
end
```

2.462 TIMER Service Subroutine

Description

Hundredths of seconds elapsed since midnight.

Syntax

```
CALL TIMER ( ix )
```

Argument(s)

ix

Default INTEGER scalar. It is assigned the hundredths of a second elapsed.

If the TIMER is not able to retrieve the elapsed time, -1 is returned.

Example

```
use service_routines,only:timer
integer :: ix
call timer(ix)
write(6,*) ix
end
```

2.463 TINY Intrinsic Function

Description

Smallest representable positive number of data type.

Class

Inquiry function.

Syntax

```
result = TINY ( X )
```

Argument(s)

X

X shall be of type REAL. It can be scalar or array-valued.

Result

The result is a scalar of the same type and kind as *X*. Its value is the smallest positive number in the data type of *X*.
The result value is as follows:

Type of *X*	The result value
Single precision REAL	1.17549435E-38_4
Double precision REAL	2.225073858507201E-308_8
Quadruple precision REAL	3.362103143112093506262677817321752 6E-4932_16

Example

```
a = tiny (4.0) ! a is assigned the value 1.17549435E-38
```

2.464 TRAILZ Intrinsic Function

Description

Number of trailing zero bits.

Class

Elemental function.

Syntax

Generic name	Specific name	Number of args	Data type of argument(s)	Data type of result
TRAILZ	---	1	INTEGER	Default INTEGER

```
result = TRAILZ ( I  )
```

Argument(s)

I

I shall be of type INTEGER.

Result

The result is of the default INTEGER type.

Remarks

TRAILZ evaluates number of trailing zero bits in *I*. If *I* is zero, the result value is BIT_SIZE(*I*).

The result is of the default INTEGER type.

Example

```
k = trailz(-1)    ! k is assigned the value 0
m = trailz (0)    ! m is assigned the value 32
n = trailz (8)    ! n is assigned the value 3
```

2.465 TRANSFER Intrinsic Function

Description

Interpret the physical representation of a number with the type and type parameters of a given number.

Class

Transformational function.

Syntax

```
result = TRANSFER ( SOURCE , MOLD [ , SIZE ] )
```

Required Argument(s)

SOURCE

SOURCE can be of any type.

MOLD

MOLD can be of any type.

Optional Argument(s)

SIZE

SIZE shall be a scalar of type INTEGER. The corresponding actual argument shall not be an optional dummy argument.

Result

The result is of the same type and type parameters as *MOLD*. If *MOLD* is a scalar and *SIZE* is absent the result is a scalar. If *MOLD* is array-valued and *SIZE* is absent, the result is array valued and of rank one. Its size is as small as possible such that it is not shorter than *SOURCE*. If *SIZE* is present, the result is array-valued of rank one and of size *SIZE*.

If the physical representation of the result is the same length as the physical representation 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 of the result is that of *SOURCE* and the trailing part is undefined. 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*.

Example

```
real :: a
integer :: i
a = transfer(i,a) ! a is assigned the physical
                  ! representation of i
```

2.466 TRANSPOSE Intrinsic Function

Description

Transpose an array of rank two.

Class

Transformational function.

Syntax

result = TRANSPOSE (*MATRIX*)

Argument(s)

MATRIX

MATRIX can be of any type. It shall be of rank two.

Result

The result is of the same type, kind, and rank as *MATRIX*. Its shape is (*n*, *m*), where (*m*, *n*) is the shape of *MATRIX*. Element (*i* , *j*) of the result has the value *MATRIX* (*j* , *i*).

Example

```
integer, dimension(2,3):: a = reshape((/1,2,3,4,5,6/),(/2,3/))
! represents the matrix |1 3 5|
!                       |2 4 6|
integer, dimension(3,2) :: b
b = transpose(a) ! b is assigned the value
!                       |1 2|
!                       |3 4|
!                       |5 6|
```

2.467 TRIM Intrinsic Function

Description

Omit trailing blanks.

Class

Transformational function.

Syntax

```
result = TRIM ( STRING )
```

Argument(s)

STRING

STRING shall be of type CHARACTER and shall be scalar.

Result

The result is of the same type and kind as *STRING*. Its value and length are those of *STRING* with trailing blanks removed.

Example

```
character(len=6) shorter
shorter = trim("longer   ")
        ! shorter is assigned the value "longer"
```

2.468 TTYNAM Service Function

Description

Returns a blank padded path name of the terminal device associated with the logical *unit*.

Syntax

```
name = TTYNAM ( unit )
```

Argument(s)

unit

Default INTEGER scalar. Unit number that is connected to a file.

Result

Default CHARACTER scalar. A blank padded path name of the terminal device associated with the logical input/output unit.

Example

```
use service_routines,only:ttynam,isatty
if (isatty(6)) print *,ttynam(6)
end
```

2.469 Type Declaration Statement

Description

The type declaration statement declares the type, type parameters, and attributes of data objects.

Syntax

```
type-spec [[,attr-spec] ... ::]entity-decl-list
```

Where:

type-spec is the declaration type specifier, and the following syntax.

```
intrinsic-type-spec        or
TYPE ( derived-type-spec )  or
CLASS( derived-type-spec )  or
CLASS( * )
```

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
BYTE
```

kind-selector is

```
( [ KIND = ] kind )         or
* mem-length
```

char-selector is

```
( LEN=char-length-parm , KIND=kind )        or
( char-length-parm , [ KIND= ]kind )        or
( KIND=kind , LEN=char-length-parm )        or
( [ LEN= ]char-length-parm)                 or
* char-length (obsolescent feature)
```

char-length is

```
( char-length-parm )              or
scalar-int-literal-constant
```

scalar-int-literal-constant is a scalar INTEGER literal constant.

char-length-parm is

```
scalar-int-expr        or
*                      or
:
```

kind is the value of kind type parameter. *kind* is a scalar INTEGER initialization expression.

mem-length is a number of bytes used to represent each respective type. It shall be a scalar INTEGER initialization expression. If the *type-spec* is INTEGER, REAL, or LOGICAL, the value of *mem-length* has the same meaning as the value of *kind*. If the *type-spec* is COMPLEX, the value of *mem-length* has the same meaning as the value of *kind**2.

scalar-int-expr is a specification expression.

A character length parameter value of ':' indicates a deferred type parameter.

The value can be specified for *kind* and *mem-length* are shown in the following table.

type-spec	*kind*	*mem-length*	Type name
INTEGER	1 2 4 or absence 8	1 2 4 or absence 8	One-byte INTEGER Two-byte INTEGER Four-byte INTEGER (default INTEGER) Eight-byte INTEGER
REAL	4 or absence 8 16	4 or absence 8 16	Single precision REAL (default REAL) Double precision REAL Quadruple precision REAL
DOUBLE PRECISION	---	---	Double precision REAL
COMPLEX	4 or absence 8 16	8 or absence 16 32	Single precision COMPLEX (default COMPLEX) Double precision COMPLEX Quadruple precision COMPLEX
LOGICAL	1 2 4 or absence 8	1 2 4 or absence 8	One-byte LOGICAL Two-byte LOGICAL Four-byte LOGICAL (default LOGICAL) Eight-byte LOGICAL

type-spec	*kind*	*mem-length*	Type name
CHARACTER	1 or absence 4	---	One-byte CHARACTER (default CHARACTER) Four-byte CHARACTER (ISO_10646 CHARACTER) (The length type parameter is specified by *char-selector*)
BYTE	---	---	One-byte INTEGER

derived-type-spec is a derived type specifier. See "1.5.11.8 Derived Type Specifier" for derived type specifier.

attr-spec is

```
ALLOCATABLE                                                    or
ASYNCHRONOUS                                                   or
AUTOMATIC                                                      or
CODIMENSION left-square-bracket  coarray-spec  right-square-bracket  or
CONTIGUOUS                                                     or
DIMENSION ( array-spec )                                       or
EXTERNAL                                                       or
INTENT ( intent-spec )                                         or
INTRINSIC                                                      or
language-binding-spec                                          or
OPTIONAL                                                       or
PARAMETER                                                      or
POINTER                                                        or
PRIVATE                                                        or
PROTECTED                                                      or
PUBLIC                                                         or
SAVE                                                           or
STATIC                                                         or
TARGET                                                         or
VALUE                                                          or
VOLATILE
```

coarray-spec is a coarray specification. See "2.73 CODIMENSION Statement" for coarray specification

left-square-bracket is a left-side square bracket. The left-side square bracket is '['.

right-square-bracket is a right-side square bracket. The right-side square bracket is ']'.

array-spec is an array specification. See "2.113 DIMENSION Statement" for array specification.

intent-spec is

```
IN       or
OUT      or
INOUT
```

language-binding-spec is

```
BIND ( C [ , NAME = scalar-char-initialization-expr ]
```

See "2.15 ALLOCATABLE Statement", "2.30 ASYNCHRONOUS Statement", "2.40 AUTOMATIC Statement", "2.45 BIND Statement", "2.73 CODIMENSION Statement", "2.82 CONTIGUOUS Statement", "2.113 DIMENSION Statement", "2.169 EXTERNAL Statement", "2.275 INTENT Statement", "2.277 INTRINSIC Statement", "2.355 OPTIONAL Statement", "2.358 PARAMETER Statement", "2.361 POINTER Statement", "2.370 PRIVATE Statement", "2.377 PROTECTED Statement", "2.378 PUBLIC Statement", "2.404 SAVE Statement", "2.442 STATIC Statement", "2.458 TARGET Statement", "2.481 VALUE Statement", or "2.483 VOLATILE Statement" for each attribute specifier.

The same *attr-spec* shall not appear more than once in a type declaration statement.

entity-decl-list is a comma-separated list of

```
object-name [ ( array-spec ) ] [ * length ] [ initialization ]         or
object-name [ * length ] [ ( array-spec ) ] [ initialization ]         or
object-name [ ( array-spec ) ]                                    &
&     [ left-square-bracket  coarray-spec  right-square-bracket  ]  &
&     [ * length ] [ initialization ]                                  or
object-name [ * length ] [ ( array-spec ) ]                       &
&     [ left-square-bracket  coarray-spec  right-square-bracket  ]  &
&     [ initialization ]                                               or
function-name [ * length ]
```

object-name is the name of a data object being declared.

length is

```
char-length        or
mem-length
```

char-spec in *entity-decl* may be present if the *type-spec* is CHARACTER. *mem-length* in *entity-decl* may be present if the *type-spec* is INTEGER, REAL, COMPLEX, or LOGICAL.

initialization is

```
= initialization-expr        or
=> NULL ( )                  or
/ data-stmt-value-list /
```

':' as a *char-length* type parameter value can be used only in the declaration of an entity or component with the POINTER or ALLOCATABLE attribute.

initialization-expr is an initialization expression.

See "2.106 DATA Statement" for *data-stmt-value-list*.

function-name is the name of a function being declared. *function-name* shall be the name of the external function, intrinsic function, dummy function, or statement function.

Remarks

A named data object shall not be explicitly specified to have a particular attribute more than once in a scoping unit.

The ALLOCATABLE attribute can be used only when declaring an array.

An array declared with a POINTER or an ALLOCATABLE attribute shall be specified with a deferred shape.

An *array-spec* for a *function-name* that does not have the POINTER and ALLOCATABLE attribute shall be an *explicit-shape-spec-list*.

If the POINTER attribute is specified, the TARGET, or INTRINSIC attribute shall not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.

The PARAMETER attribute shall not be specified for dummy arguments, pointers, allocatable variables, derived type objects with an ultimate component that is an allocatable variable, functions, or objects in a common block.

The INTENT, OPTIONAL, and VALUE attributes may be specified only for dummy arguments.

An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attributes specified unless it is a function.

The = *initialization-expr* shall appear if the statement contains a PARAMETER attribute.

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

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 array, a derived type object containing an ultimate component that is an allocatable array, an external name, an intrinsic name, or an automatic object.

If =>NULL( ) appears in *initialization*, the object shall have the POINTER attribute.

If = *initialization-expr* or / *data-stmt-value-list* / appears in *initialization*, the object shall not have the POINTER attribute.

If the VOLATILE attribute is specified, the PARAMETER, INTRINSIC, or EXTERNAL attribute shall not be specified.

If the VALUE attribute is specified, the PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, INTENT(INOUT), or INTENT(OUT) attribute shall not be specified.

When the VALUE attribute is specified, the value of character length parameter must be omitted or specified in an initialization expression.

The VALUE attribute must not be specified in a dummy procedure.

A language binding specifier can be written in only the module declaration part.

An entity declared by a language binding specifier must be an interoperable variable.

If a language binding specifier is specified with a NAME= specifier, the entity declarations list must be only one entity declaration.

The PROTECTED attribute can be specified only in specification part of a module.

The PROTECTED attribute can be specified only in named variable excluding a common block.

When the PROTECTED attribute is specified, an EXTERNAL attribute, an INTRINSIC attribute or a PARAMETER attribute must not be specified.

When a non-pointer object that has a PROTECTED attribute is referenced as association, the object must not be appeared in a statement that a variable defines or undefines and data target in pointer assignment.

When a pointer object that has the PROTECTED attribute is referenced as use association, the object must not be appeared in the following items.

- A pointer object in a NULLIFY statement
- A data pointer object in a pointer assignment statement
- An allocatable object in an ALLOCATE statement or a DEALLOCATE statement
- An actual argument when a pointer that has an INTENT(INOUT) or an INTENT(OUT) attribute referenced a procedure in a dummy argument.

The presence of *initialization* implies that *object-name* is saved, for an *object-name* in a named common block or an *object-name* with the PARAMETER attribute.

An internal function name, a module function name, and function of interface body shall not be declared with an asterisk *char-length-parm-value*.

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

The CLASS specifier can be used to declare a polymorphic object. If the CLASS specifier includes a type name, that type becomes the declared type of the polymorphic object.

An object declared with the CLASS(*) specifier is unlimited polymorphic. An unlimited polymorphic data entity does not have a declared type. It is treated as having a declared type different to that of all other entities.

An entity declared using CLASS must be a dummy argument or must have the ALLOCATABLE or POINTER attribute. It cannot have the VALUE attribute.

The CONTIGUOUS attribute can be specified only with pointer array or assumed shape array.

Example

```
integer :: a,b,c          ! a, b, and c are of type integer
real, dimension(2,4) :: d
                          ! d is a 2 by 4 array of real
integer :: e = 2          ! integer e is initialized
```

2.470 TYPE Statement (Derived Type Definition)

Description

This TYPE statement begins a derived type definition. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
TYPE [[, type-attr-spec-list ]::] type-name[( type-param-name-list )]
```

type-attr-spec-list is

```
access-spec                     or
EXTENDS ( parent-type-name )    or
ABSTRACT                        or
BIND ( C )
```

access-spec is

```
PUBLIC       or
PRIVATE
```

Where:

type-name is the name of the derived type being defined.

Remarks

access-spec is permitted only if the derived type definition is within the specification part of a module.

parent-type-name can specify a parent type name to be inherited.

type-name shall not be the name of an intrinsic type nor of another accessible derived type name.

type-param-name-list can specify a list of type parameter name.

Example

```
type coordinates
  real :: x , y = 40.0  ! default initialization for y specified
end type coordinates
```

2.471 TYPE Type Declaration Statement

Description

The TYPE type declaration statement declares entities of derived type *type-name*. See "2.469 Type Declaration Statement" for type declaration statement.

Syntax

```
TYPE ( type-name )[[, attr-spec ] ... ::]entity-decl-list
```

2.472 TYPE IS Statement

Description

The TYPE IS statement shall be specified in a SELECT TYPE construct. See "2.413 SELECT TYPE Construct" for SELECT TYPE construct.

Syntax

```
TYPE IS ( type-spec ) [ select-construct-name ]
```

Where:

type-spec is a type specifier that is either an intrinsic type specifier or a derived type specifier. See "2.469 Type Declaration Statement" for intrinsic type specifier, and see "1.5.11.8 Derived Type Specifier" for derived type specifier.

type-spec must specify that each length type parameter is inherited.

type-spec cannot specify a sequential derived type or a type having the BIND attribute.

select-construct-name is an optional name assigned to the SELECT TYPE construct.

See "2.413 SELECT TYPE Construct" for SELECT TYPE construct.

2.473 UBOUND Intrinsic Function

Description

Upper bounds of an array or a dimension of an array.

Class

Inquiry function.

Syntax

```
result = UBOUND ( ARRAY [ , DIM , KIND ] )
```

Required Argument(s)

ARRAY

ARRAY can be of any type. It shall not be a scalar and shall not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Argument(s)

DIM

DIM shall be of type INTEGER and shall be a dimension of *ARRAY*.

KIND

KIND shall be of type INTEGER initialization expression.

Result

The result is of type INTEGER. If *KIND* is specified, the kind type parameter of the function result is following to *KIND*. If *KIND* is omitted, the kind type parameter of the function result is default INTEGER.

If *DIM* is present, the result is a scalar with the value of the upper bound of *ARRAY*. If *DIM* is absent, the result is an array of rank one with values corresponding to the upper bounds of each dimension of *ARRAY*.

The result is zero for zero-sized dimensions.

Example

```
integer, dimension(3,-4:0) :: i
integer :: k, j(2)
j = ubound (i)     ! j is assigned the value [3,0]
k = ubound (i, 2)  ! k is assigned the value 0
```

2.474 UCOBOUND Intrinsic Function

Description

Returns upper cobound(s) of a coarray.

Class

Inquiry function.

Syntax

```
result = UCOBOUND ( COARRAY [, DIM, KIND ] )
```

Required Argument(s)

COARRAY

COARRAY must be a coarray. If it is allocatable, it must be allocated.

Optional Argument(s)

DIM

DIM must be an integer scalar with a value in the range of 1 and n, where n is the corank of *COARRAY*. The corresponding actual argument must not be an optional dummy argument.

KIND

KIND must be a scalar integer initialization expression.

Result

It is of type integer. If *KIND* is specified, its kind type parameter is the value of *KIND*. Otherwise, the kind type parameter is default integer. If *DIM* is specified, *result* is defined with the value of upper cobound for cosubscript *DIM* of *COARRAY*. Otherwise it is an array of rank one and size n, where n is the corank of *COARRAY*. In that case, the value of each element of *result* is each upper cobound of the corresponding cosubscript.

Example

```
integer, save :: k[2,3:4,*]
if (num_images()==10) then
  print *,ucobound(k,dim=1) ! 2 is output
  print *,ucobound(k,dim=2) ! 4 is output
  print *,ucobound(k,dim=3) ! 3 is output
end if
```

2.475 UNION Statement

Description

The UNION statement begins a union declaration in derived type definition by STRUCTURE statement. See "1.5.11.1 Derived Type Definition" for derived type definition.

Syntax

```
UNION
```

Example

```
structure /complex_element/
  union
    map
      real :: real,imag
    end map
    map
      complex :: complex
    end map
  end union
end structure
record /complex_element/ x
x%real = 2.0
x%imag = 3.0
print *,x%complex     ! complex has the value (2.0,3.0)
```

2.476 UNLINK Service Function

Description

Removes a specified directory entry.

Syntax

```
iy = UNLINK ( name )
```

Argument(s)

name

Default CHARACTER scalar. Path of the directory to delete.

Its length shall be less than or equal to MAXPATHLEN as defined in the system include.

Result

Default INTEGER scalar. Zero if successful; otherwise, a system error code.

Example

```
use service_routines,only:unlink
integer :: iy
iy = unlink('test/xx')
end
```

2.477 UNLOCK Statement

Description

The UNLOCK statement unlocks lock variable.

Syntax

`UNLOCK ( lock-variable [ , sync-stat-list ] )`

Where:

lock-variable is a lock variable. It must be of type LOCK_TYPE. See "1.18.3 LOCK_TYPE Type" for LOCK_TYPE.

sync-stat-list is a comma-separated list of

```
STAT = stat-variable            or
ERRMSG = errmsg-variable
```

See "1.18.2 Synchronization Status Specifier" for *stat-variable* and *errmsg-variable*.

Remarks

Lock variable is locked by an image if it was locked by the image and it has not been unlocked by the image.

Execution of UNLOCK statement unlocks lock variable.

If a lock variable is unlocked by the execution of UNLOCK statement on image M and then locked by the execution of LOCK statement on image T, segments preceding the UNLOCK statement on image M precede segments following the LOCK statement on image T.

If a lock variable in UNLOCK statement is not locked by the executing image, an error condition occurs. If an error condition occurs during the execution of UNLOCK statements, the value of the lock variable is not changed.

Example

```
use iso_fortran_env, only : lock_type
type(lock_type) :: x[*]
if (this_image() == 4) then
    lock(x)     ! lock variable "x" is locked by image 4
    unlock(x)   ! lock variable "x" is unlocked by image 4
end if
end
```

2.478 UNPACK Intrinsic Function

Description

Unpack an array of rank one into an array under control of a mask.

Class

Transformational function.

Syntax

```
result = UNPACK ( VECTOR , MASK , FIELD )
```

Argument(s)

VECTOR

VECTOR can be of any type. It shall be of rank one. Its size shall be at least as large as the number of true elements in *MASK*.

MASK

MASK shall be of type LOGICAL. It shall be array-valued.

FIELD

FIELD shall be of the same type and type parameters as *VECTOR*. It shall be conformable with *MASK*.

Result

The result is an array of the same type and type parameters as *VECTOR* and the same shape as *MASK*. The element of the result that corresponds to the *i*th element of *MASK*, in array-element order, has the value *VECTOR*(*i*) for *i* =1,2,...,*t* , where *t* is the number of true values in *MASK*. Each other element has the value *FIELD* if *FIELD* is scalar or the corresponding element in *FIELD*, if *FIELD* is an array.

Example

```
integer, dimension(9) :: c = (/0,3,2,4,3,2,5,1,2/)
logical, dimension(2,2) :: d
integer, dimension(2,2) :: e
d = reshape((/.false.,.true.,.true.,.false./),(/2, 2/))
e = unpack(c,mask=d,field=-1)
! e is assigned |-1  3|
!               | 0 -1|
```

2.479 USE Statement

Description

The USE statement specifies that a specified module is accessible by the current scoping unit. It also provides a means of renaming or limiting the accessibility of entities in the module.

Syntax

```
USE [[, module-nature ] :: ]module-name[, rename-list ]          or
USE [[, module-nature ] :: ]module-name, ONLY : [ only-list ]
```

Where:

module-nature is

```
INTRINSIC           or
NON_INTRINSIC
```

module-nature is the nature of module.

module-name is the name of a module.

rename-list is a comma-separated list of

```
local-name => use-name            or
OPERATOR ( local-defined-operator ) => OPERATOR ( use-defined-operator )
```

local-name is the local name for the entity specified by *use-name*.

use-name is the name of an entity in the specified module.

local-defined-operator is

```
defined-unary-operator      or
defined-binary-operator
```

local-defined-operator is a local defined operator.

use-defined-operator is

```
defined-unary-operator      or
defined-binary-operator
```

use-defined-operator is a use defined operator.

defined-unary-operator is a defined unary operator. *defined-binary-operator* is a defined binary operator.

Both forms are as follows.

```
. operator-name .
```

operator-name is a user-defined name for the operation, consisting of one to 240 letters.

only-list is a comma-separated list of

```
generic-spec         or
only-use-name        or
only-rename
```

generic-spec is a generic specifier and cannot be a generic binding. It has the following format:

```
generic-name                    or
OPERATOR (defined-operator)     or
ASSIGNMENT ( = )                or
dtio-generic-spec
```

generic-name is the name of a generic procedure.

defined-operator cannot be a generic binding. It has the following format:

```
intrinsic-operator        or
. operator-name .
```

operator-name is a user-defined name for the operation, consisting of one to 240 letters.

ASSIGNMENT (=) is a user-defined assignment.

dtio-generic-spec is a derived type input/output procedure that is the following syntax:

```
READ ( FORMATTED )        or
READ ( UNFORMATTED )      or
WRITE ( FORMATTED )       or
WRITE ( UNFORMATTED )
```

only-use-name is

```
use-name
```

only-rename is

```
local-name => use-name
```

Remarks

A module name shall be a name of intrinsic module if the module nature is INTRINSIC.

A module name shall be a name of non intrinsic module if the module nature is NON_INTRINSIC.

The intrinsic module, or non intrinsic module that has the same name shall not be referenced in the scoping unit of a module.

Each *use-defined-operator*, *use-name* and *generic-spec* shall be public entity in the module.

The USE statement that has no *module-nature* provides enables either of intrinsic module or non intrinsic module. The accessible module is a non intrinsic module if the module name is the same name of both intrinsic module and non intrinsic module.

A USE statement without ONLY provides access to all PUBLIC entities in the specified module.

A USE statement with ONLY provides access only to *generic-spec*, *use-name* and *use-defined-operator* that appear in the *only-list*.

More than one USE statement for a given module may appear in a scoping unit. If one of the USE statements is without an ONLY qualifier, all public entities in the module are accessible. If all the USE statements have ONLY qualifiers, only those entities named in one or more of the *only-list*s are accessible.

An accessible entity in the referenced module has one or more local names. These names are

- The name of the entity in the referenced module if that name appears as an *only-use-name* in any *only* for that module,
- Each of the *local-name*s the entity is given in any *rename* or *only-rename*, or *local-defined-operator* for that module, and
- The name of the entity in the referenced module if that name does not appear as a *use-name* in any *rename*, *only-rename* or *use-defined-operator* for that module.

Two or more accessible entities, other than generic interfaces or defined operator, may have the same name only if the name is not used to refer to an entity. If two or more generic interfaces or defined operators that are accessible in the same scoping unit have the same name, same operator, or are assignments, they are interpreted as a single generic interface. Except for these cases, the local name of any entity given accessibility by a USE statement shall differ from the local names of all other entities accessible to the scoping unit through USE statements and otherwise.

An entity can be accessed by more than one *local-name*.

A *local-name* shall not be respecified with differing attributes in the scoping unit that contains the USE statement, except that it can appear in a PUBLIC or PRIVATE statement in the scoping unit of a module.

Even if an entity within a use associated module has neither the ASYNCHRONOUS attribute nor the VOLATILE attribute, it can have the ASYNCHRONOUS attribute or the VOLATILE attribute in a local scope.

Example

```
module types
  type pair_int
    integer :: i1,i2
  end type pair_int
  type pair_real
    real :: r1,r2
  end type pair_real
end module types
module pair_data
  use types             ! use all public entities in types
  type(pair_int) :: ints_data
  type(pair_real) :: reals_data
end module pair_data
use types, ints => pair_int
                        ! use all public entities in types, and
                        ! refer to pair_int as ints locally
use pair_data , only : ints_data
                        ! use only ints_data from pair_data
```

2.480 VAL Intrinsic Function

Description

Pass an item to a procedure by value. VAL can only be used as an actual argument.

Class

Utility function.

Syntax

```
CALL subroutine-name ( VAL ( X ) )
function-name ( VAL ( X ) )
```

Argument(s)

X

X shall be a scalar expression of type one-byte INTEGER, two-byte INTEGER, four-byte INTEGER, eight-byte INTEGER, one-byte LOGICAL, two-byte LOGICAL, four-byte LOGICAL, eight-byte LOGICAL, single precision REAL, or double precision REAL.

Result

VAL gives the actual argument as an immediate value.

This function can only be used on the actual argument of procedure reference.

Example

```
i = my_c_function(val(a)) ! a is passed by value
```

2.481 VALUE Statement

Description

The VALUE statement directs the dummy argument to be associated with a definable temporary area whose initial value is that of the actual argument. Neither the value nor the changes in the definition status for the dummy argument influence the actual argument.

Syntax

```
VALUE [ :: ] dummy-arg-name-list
```

Where:

dummy-arg-name-list is a comma-separated list of the name of dummy arguments.

Remarks

dummy-arg-name shall not be the name of dummy procedure, dummy pointer, and dummy array.

The scalar dummy argument that has the VALUE attribute is of type one-byte INTEGER, two-byte INTEGER, four-byte INTEGER, eight-byte INTEGER, one-byte LOGICAL, two-byte LOGICAL, four-byte LOGICAL, eight-byte LOGICAL, single precision REAL, double precision REAL or CHARACTER whose character length parameter is constant expression.

Example

```
subroutine csub(i)
  value :: i  ! The dummy argument "i" is associated with
              ! a definable temporary area which initial
              ! value is that of the actual argument
```

2.482 VERIFY Intrinsic Function

Description

Verify that a set of characters contains all the characters in a string.

Class

Elemental function.

Syntax

```
result = VERIFY ( STRING , SET [ , BACK , KIND ] )
```

Required Argument(s)

STRING

STRING shall be of type CHARACTER.

SET

SET shall be of the same kind and type as *STRING*.

Optional Argument(s)

BACK

BACK shall be of type LOGICAL.

KIND

KIND shall be of type INTEGER initialization expression.

Result

The result is of type INTEGER. If *KIND* is specified, the kind type parameter of the function result is following to *KIND*. If *KIND* is omitted, the kind type parameter of the function result is default INTEGER.

If *BACK* is absent, or if it is present with the value false, the value of the result is the position number of the leftmost character in *STRING* that is not in *SET*. If *BACK* is present with the value true, the value of the result is the position number of the rightmost character in *STRING* that is not in *SET*. The value of the result is zero if each character in *STRING* is in *SET*, or if *STRING* has length zero.

Example

```
i = verify ("Lalalalala","l") ! i is assigned the
                              ! value 1
i = verify ("LalalaLALA","LA",back=.true.)
                              ! i is assigned the
                              ! value 6
```

2.483 VOLATILE Statement

Description

The VOLATILE statement declares specified objects are prevented optimization.

Syntax

```
VOLATILE [ :: ] object-name-list
```

Where:

object-name-list is a comma-separated list of an object name.

object-name shall not be a named constant or an intrinsic procedure.

Example

```
program main
  volatile :: v,w     ! v and w are prevented optimization
```

2.484 WAIT Service Function

Description

Causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last wait, return is immediate; if there are no children, return is immediate with an error code.

Syntax

```
iy = WAIT ( status )
```

Argument(s)

status

Default INTEGER scalar. The child process' termination status.

Result

Default INTEGER scalar. If the returned value is positive, it is the process ID of the child and status is its termination status. If the returned value is negative, it is the negation of a system error code.

Example

```
      use service_routines,only:wait,fork
      integer :: pid,stat
      pid = fork()
      if (pid>0) go to 30
      call sub
   30 if (wait(stat).eq.-1) write(*,*) 'wait error'
      call sub1
      end
```

2.485 WAIT Statement

Description

The WAIT statement performs a wait operation for an asynchronous data transfer operation.

Syntax

```
WAIT ( wait-spec-list )
```

Where:

wait-spec-list is the wait specifier list, and has the format below:

```
[ UNIT = ] external-file-unit      or
END = end-label                    or
EOR = eor                          or
ERR = err-label                    or
ID = id                            or
IOMSG = iomsg                      or
IOSTAT = io-stat
```

Remarks

Each specifier can be specified only once in the wait specifier list.

external-file-unit is an external file device and must be a scalar INTEGER expression.

The file device number must be specified. If the optional character string UNIT= is omitted, the file device number must be the first item in the wait specifier list.

io-stat must be a scalar INTEGER variable.

If the IOSTAT= specifier is specified, the following values are set in the variable in the IOSTAT= specifier:

- If no error condition, file end condition, or record end condition is detected: 0
- If an error condition is detected: 1 or the execution-time diagnostic message number
- If a file end condition is detected: -1
- If a record end condition is detected: -2

err-label is the statement number, and must be the statement number of a jump destination statement within the same valid range as this WAIT statement.

If the ERR= specifier is specified and an error condition is detected during WAIT statement execution, the statement with the statement number specified in the ERR= specifier is executed next.

end-label must be the statement number of a jump destination statement within the same valid range as this WAIT statement.

If the END= specifier is specified and a file end condition is detected during WAIT statement execution, the statement with the statement number specified in the END= specifier is executed next.

eor-label must be the statement number of a jump destination statement within the same valid range as this WAIT statement.

If the EOR= specifier is specified and a record end condition is detected during WAIT statement execution, the statement with the statement number specified in the EOR= specifier is executed next.

The value of the expression specified in the ID= specifier must be a data transfer operation identifier at the specified device. If the ID= specifier is specified, the wait operation is performed for the specified data transfer operation. If the ID= specifier is omitted, the wait operation is performed for all data transfer operations at the device.

iomsg must be a scalar default CHARACTER variable. If the following condition occurs during input/output statement execution, a message for explanation purposes is assigned to *iomsg*:

- Error condition
- File end condition
- Record end condition

In other cases, the *iomsg* value does not change.

If it is not present or not connected to a file, or if the specified device was not opened as asynchronous input/output, the WAIT statement can still be executed. The WAIT statement will be executed in the same way as one in which the ID= specifier is omitted. An error condition or file end condition is not possible for this type of WAIT statement.

2.486 WHERE Construct

Description

The WHERE construct controls which elements of an array will be affected by a block of assignment statements. This is also known as masked array assignment.

Syntax

```
[ where-construct-name : ] WHERE ( mask-expr )
    [ where-body-construct ] ...
[ ELSEWHERE ( mask-expr) [ where-construct-name ]
    [ where-body-construct ] ... ] ...
[ ELSEWHERE [ where-construct-name ]
    [ where-body-construct ] ... ]
END WHERE [ where-construct-name ]
```

Where:

where-construct-name is an optional name given to the WHERE construct.

If the WHERE construct statement is identified by *where-construct-name*, the corresponding END WHERE statement shall specify the same *where-construct-name*. If the WHERE construct statement is not identified by *where-construct-name*, the corresponding END WHERE statement shall not specify a *where-construct-name*. If a masked ELSEWHERE statement or ELSEWHERE statement is identified by a *where-construct-name*, the corresponding WHERE construct statement shall specify the same *where-construct-name*.

mask-expr is an array LOGICAL expression.

where-body-construct is

```
assignment-stmt      or
WHERE statement      or
WHERE construct
```

Remarks

An ELSEWHERE statement that has a *mask-expr* is a masked ELSEWHERE statement.

If a WHERE construct contains a WHERE statement, a masked ELSEWHERE statement or another WHERE construct then each *mask-expr* within the WHERE construct shall have the same shape.

The variable being defined in assignment statement that appears as a *where-body-construct* shall have the same shape as *mask-expr*.

An assignment statement that appears as a *where-body-construct* is a defined assignment shall be elemental.

When a WHERE construct statement is executed, a control mask is established to be the value of *mask-expr*, and a pending control mask is established to have the value .NOT. *mask-expr*.

Each statement in a WHERE construct is executed in sequence.

Upon execution of a masked ELSEWHERE statement, the control mask is established to have the value (the pending control mask immediately before) .AND. *mask-expr*, and a pending control mask is established to have the value .NOT. (new control mask).

Upon execution of ELSEWHERE statement, the control mask is established to have the value of the pending control mask immediately before.

Upon execution of a WHERE statement or WHERE construct statement that is part of a *where-body-construct*, the control mask is established to have the value (the control mask immediately before) .AND. *mask-expr*, and a pending control mask is established to have the value .NOT. (new control mask).

Upon execution of an END WHERE statement, the control mask and pending control mask are established to have the values they had prior to the execution of the corresponding WHERE construct statement. Following the execution of a WHERE statement that appears as a *where-body-construct*, the control mask is established to have the value it had prior to the execution of the WHERE statement.

When assignment statement that appears as a *where-body-construct* is executed, the right-hand side of the assignment and the expression in the left-hand side of the assignment are evaluated for all elements where *mask-expr* is true and the result assigned to the corresponding elements of the left-hand side.

Example

```
integer,dimension(10):: a,b,c
a = (/ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 /)
b = (/-5,-3,-1, 1, 3, 5, 7, 9,11,13 /)
where (a>b)
  c = a
elsewhere (a==b)
  c = 0
elsewhere
  c = b
end where
           ! c has the value (/0,1,2,3,4,0,7,9,11,13/)
```

2.487 WHERE Construct Statement

Description

The WHERE construct statement begins a WHERE construct. See "2.486 WHERE Construct" for WHERE construct.

Syntax

```
[ where-construct-name : ] WHERE ( mask-expr )
```

Where:

where-construct-name is an optional name given to the WHERE construct.

mask-expr is an array LOGICAL expression.

2.488 WHERE Statement

Description

The WHERE statement is used to mask the assignment of values in an array assignment statement.

Syntax

```
WHERE ( mask-expr ) where-assignment-stmt
```

Where:

mask-expr is an array LOGICAL expression.

where-assignment-stmt is an assignment statement that is the following syntax:

```
assignment-stmt
```

Remarks

The variable being defined in *where-assignment-stmt* shall have the same shape as *mask-expr*.

where-assignment-stmt that is a defined assignment shall be elemental.

When *where-assignment-stmt* is executed, the right-hand side of the assignment and the expression in the left-hand side of the assignment are evaluated for all elements where *mask-expr* is true and the result assigned to the corresponding elements of the left-hand side.

Example

```
integer,dimension(10):: a,b,c=0
a = (/ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 /)
b = (/-5,-3,-1, 1, 3, 5, 7, 9,11,13 /)
where (a>b) c = a
          ! c has the value (/0,1,2,3,4,0,0,0,0,0/)
```

2.489 WRITE Statement

Description

The WRITE statement transfers values to a file from the entities specified in an output list or a namelist group.

Syntax

```
WRITE ( io-control-spec-list ) [ output-item-list ]
```

Where:

io-control-spec-list is a comma-separated list of

```
[ UNIT = ] io-unit                or
[ FMT = ] format                  or
[ NML = ] namelist-group-name     or
REC = record-number               or
IOSTAT = io-stat                  or
ERR = err-label                   or
ADVANCE = advance                 or
NUM = record-len                  or
ASYNCHRONOUS = asynchronous       or
DECIMAL = decimal                 or
DELIM = delim                     or
ID = id                           or
IOMSG = iomsg                     or
POS = pos                         or
ROUND = round                     or
SIGN = sign
```

An *io-unit* shall be specified. If the optional characters UNIT= are omitted from the *io-unit* specifier, the *io-unit* specifier shall be the first item in the *io-control-spec-list*.

io-unit is

```
external-file-unit      or
*                       or
internal-file-unit
```

external-file-unit is a scalar INTEGER expression that evaluates to the input/output unit number of an external file.

internal-file-unit is a default CHARACTER variable that is to the input/output unit file of an internal file.

If the *io-unit* specifier specifies an internal file, the REC= specifier or POS= specifier shall not be specified.

io-control-spec-list shall not contain both a *format* and a *namelist-group-name*.

If the optional characters FMT= are omitted from the *format* specifier, the *format* specifier shall be the second item in the *io-control-spec-list* and the first item shall be the *io-unit* specifier without the optional characters UNIT=.

format is

```
default-char-expr              or
label                          or
*                              or
scalar-default-int-variable
```

default-char-expr is a default CHARACTER expression that evaluates to *format-specification*. See "1.8.1 Format Specification" for format specification.

label is a statement label of a FORMAT statement in the same scoping unit as the WRITE statement.

scalar-default-int-variable is a scalar default INTEGER variable that was assigned the label only by an ASSIGN statement of a FORMAT statement in the same scoping unit.

If the optional characters NML= are omitted from the *namelist-group-name* specifier, the *namelist-group-name* specifier shall be the second item in the *io-control-spec-list* and the first item shall be the *io-unit* specifier without the optional characters UNIT=.

namelist-group-name is the name of a namelist group.

If the *namelist-group-name* is present, the *output-item-list* shall not be present.

record-number is a scalar INTEGER expression that is to the number of the direct access record that is to be written.

If the REC= specifier is present, a *namelist-group-name* or a POS= specifier shall not appear, and *format* shall not be an asterisk indicating list-directed I/O.

io-stat is a scalar INTEGER variable that is assigned 1 or a positive value that is the number of the error message generated at runtime if an error condition occurs, and zero otherwise.

err-label is a statement label of a branch target statement that appears in the same scoping unit as the WRITE statement. If an error condition occurs during execution of the WRITE statement that contains an ERR= specifier, execution continues with the statement specified in the ERR= specifier.

advance is a scalar default CHARACTER expression that evaluates to NO if non-advancing input/output is to occur, and YES if advancing input/output is to occur. The default value is YES.

An ADVANCE= specifier may be present only in a formatted sequential input/output statement with explicit format specification whose control information list does not contain an *internal-file-unit* specifier.

record-len is a scalar default INTEGER variable that is assigned the record length in units of bytes that is actually transferred by executing an unformatted output statement.

If the NUM= specifier is present, a *format* and a *namelist-group-name* shall not appear.

asynchronous must be a scalar default CHARACTER.

The ASYNCHRONOUS= specifier specifies whether an input/output statement is synchronous or asynchronous. If 'YES' is specified, the statement and input/output operation are asynchronous. If 'NO' is specified or if it is omitted, the statement and input/output operation are synchronous.

decimal must be a scalar default CHARACTER expression. The *decimal* value must be either 'COMMA' or 'POINT'. It specifies the value of the current decimal editing mode at this connection. If this specifier is omitted , the mode is not changed.

delim is a scalar default CHARACTER expression. The *delim* value must be any one of the 'APOSTROPHE', 'QUOTE', or 'NONE'. It specifies the value of the current delimiter mode on the connection. If it is omitted, the mode is not changed.

id must be a scalar INTEGER variable. When an asynchronous data transfer statement with an ID= specifier is successfully executed, the variable specified by the ID= specifier is defined. This value can be used by subsequent WAIT statements or INQUIRE statements to identify a particular data transfer operation.

If an error occurs during execution of the data transfer statement, the variable specified in the ID= specifier becomes undefined.

iomsg must be a scalar default CHARACTER variable. If the following conditions occur during input/output statement execution, an explanatory message is assigned to *iomsg*:

- Error condition
- File end condition
- Record end condition

In other cases, the *iomsg* value does not change.

pos must be a scalar INTEGER variable. POS specifies a file position within a file storage unit. It can be specified in a data transfer statement only if a device connected as a stream access is specified.

round must be a scalar default CHARACTER expression. The round value is one of the values prescribed by the OPEN statement ROUND= specifier. The ROUND= specifier temporarily changes the input/output rounding mode of that connection. If ROUND is omitted, the mode is not changed.

sign must be a scalar default CHARACTER expression. The value of *sign* is either 'PLUS', 'SUPPRESS', or 'PROCESSOR_DEFINED'. The SIGN= specifier temporarily changes the sign mode of that connection. If SIGN is omitted, the mode is not changed.

output-item-list is a comma-separated list of

```
expr              or
io-implied-do
```

expr is an expression.

io-implied-do is

```
( output-item-list , io-implied-do-control )
```

io-implied-do-control is

```
do-variable = scalar-expr , scalar-expr [ , scalar-expr ]
```

do-variable is a named scalar variable of type INTEGER, default REAL, or double precision REAL. The *do-variable* of type default REAL or double precision REAL is deleted feature.

scalar-expr is a scalar expression of type INTEGER, default REAL, or double precision REAL. The *scalar-expr* is of type default REAL or double precision REAL is deleted feature.

Remarks

For an implied-DO, the loop initialization and execution is the same as for a DO construct (see "2.114 DO Construct").

The *do-variable* of an *io-implied-do-control* that is contained within another *io-implied-do* shall not appear as the *do-variable* of the containing *io-implied-do*.

Output items cannot be polymorphic unless they are processed by a user-defined derived type input/output procedure.

If an output item is a pointer, it shall be currently associated with a target and data are transferred from the target to the file.

If an output item is an allocatable variable, it shall be currently allocated.

If an array appears as an output item, it is treated as if the elements are specified in array-element order (see "1.5.8.3 Array Element Order").

If a derived type has a pointer component or an allocatable component as the ultimate component, this derived type object must be processed by a user-defined derived type input/output procedure.

If not processed by a user-defined derived type input/output procedure and if a derived type object name is specified in the output list, all those components are handled as being specified in the same order as in the derived type definition.

Example

```
write (*,*) a,b,c ! write a, b, and c using list-
                  ! directed i/o
write (3, fmt= "(e7.4)") x
                  ! write x to unit 3 using e format
write (*,10) i,j,k
                  ! write i, j, and k using format on
                  ! line 10
```

Appendix A Intrinsic Procedures

The tables below offer a synopsis of intrinsic procedures. For detailed information on individual procedures, see "Chapter 2 Alphabetical Reference".

All procedures in these tables are intrinsic. Specific function names may be passed as actual arguments except for where indicated by an asterisk in the tables. Note that for almost all programming situations it is best to use the generic procedure name.

Table A.1 Numeric Functions

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Conversion to INTEGER type y=int(x) (*3)	INT (A[,KIND])	-- -- -- -- INT IFIX JINT JIFIX IDINT JIDINT IQINT -- -- -- --	1	I1 I2 I4 I8 R R R R D D Q C DC QC nd					I4 I4 I4 I4 I4 I4 I4 I4 I4 I4 I4 I4 I4 I4 I4	Elemental function
		-- -- -- -- -- -- -- -- -- -- --	2	I1 I2 I4 I8 R D Q C DC QC nd	i i i i i i i i i i i				i (*1) i (*1) i (*1) i (*1) i (*1) i (*1) i (*1) i (*1) i (*1) i (*1) i (*1)	
	INT4	-- -- -- -- -- -- -- -- -- --	1	I1 I2 I4 I8 R D Q C DC QC					I4 I4 I4 I4 I4 I4 I4 I4 I4 I4	
	JFIX	-- -- -- -- -- -- -- --	1	I1 I2 I4 I8 R D Q C					I4 I4 I4 I4 I4 I4 I4 I4	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
		-- --		DC QC					I4 I4	
Conversion to one-byte INTEGER type y=int(x) (*3)	INT1	-- -- -- -- -- -- -- -- -- --	1	I1 I2 I4 I8 R D Q C DC QC					I1 I1 I1 I1 I1 I1 I1 I1 I1 I1	Elemental function
Conversion to two-byte INTEGER type y=int(x) (*3)	INT2	HFIX IINT IIFIX IIDINT	1	R R R D					I2 I2 I2 I2	Elemental function
		-- -- -- -- -- -- -- -- -- --	1	I1 I2 I4 I8 R D Q C DC QC					I4 I4 I4 I4 I4 I4 I4 I4 I4 I4	
Conversion to REAL type (*4)	REAL (A[,KIND])	-- -- REAL -- FLOATI FLOAT FLOATJ -- SNGL SNGLQ -- -- -- --	1	I1 I2 I4 I8 I2 I4 I4 R D Q C DC QC nd					R R R R R R R R R R R D Q R	Elemental function
		-- -- -- -- -- -- -- -- -- -- --	2	I1 I2 I4 I8 R D Q C DC QC nd	i i i i i i i i i i i				r (*1) r (*1) r (*1) r (*1) r (*1) r (*1) r (*1) r (*1) r (*1) r (*1) r (*1)	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Conversion to double-precision REAL type (*5)	DBLE (A)	-- -- DFLOTI DFLOAT DFLOTJ -- DBLE -- DBLEQ -- DREAL -- --	1	I1 I2 I2 I4 I4 I8 R D Q C DC QC nd					D D D D D D D D D D D D D	Elemental function
Conversion to quadruple-precision REAL type	QEXT (A)	-- -- QFLOAT -- QEXT QEXTD -- -- -- QREAL	1	I1 I2 I4 I8 R D Q C DC QC					Q Q Q Q Q Q Q Q Q Q	Elemental function
Conversion to COMPLEX type (*6)	CMPLX (X[,Y][,KIND])	-- -- -- -- -- -- -- --	1 or 2	I1 I2 I4 I8 R D Q nd	in in in in in in in in				C C C C C C C C	Elemental function
		-- -- -- -- -- -- -- --	2 or 3	I1 I2 I4 I8 R D Q nd	in in in in in in in in	i i i i i i i i			c (*1) c (*1) c (*1) c (*1) c (*1) c (*1) c (*1) c (*1)	
	CMPLX (X[,KIND])	-- -- --	1	C DC QC					C C C	
		-- -- --	2	C DC QC	i i i				c (*1) c (*1) c (*1)	
Conversion to double-precision COMPLEX type (*7)	DCMPLX (X[,Y])	-- -- -- -- -- -- --	1 or 2	I1 I2 I4 I8 R D Q	ir ir ir ir ir ir ir				DC DC DC DC DC DC DC	Elemental function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
		-- -- --	1	C DC QC					DC DC DC	
Conversion to quadruple-precision COMPLEX type	QCMPLX (X[,Y])	-- -- -- -- -- -- --	1 or 2	I1 I2 I4 I8 R D Q	ir ir ir ir ir ir ir				QC QC QC QC QC QC QC	Elemental function
		-- -- --	1	C DC QC					QC QC QC	
Truncation y=int(x) (*3)	AINT (A[,KIND])	AINT DINT QINT	1	R D Q					R D Q	Elemental function
		-- -- --	2	R D Q	i i i				r (*1) r (*1) r (*1)	
Rounding off y=int(x+0.5) if x >= 0 y=int(x-0.5) if x < 0 (*3)	ANINT (A[, KIND])	ANINT DNINT QNINT	1	R D Q					R D Q	Elemental function
		-- -- --	2	R D Q	i i i				r (*1) r (*1) r (*1)	
Rounding off with conversion to INTEGER y=int(x+0.5) if x >= 0 y=int(x-0.5) if x < 0 (*3)	NINT (A[,KIND])	NINT JNINT IDNINT JIDNNT IQNINT	1	R R D D Q					I4 I4 I4 I4 I4	Elemental function
		-- -- --	2	R D Q	i i i				i (*1) i (*1) i (*1)	
	I2NINT	-- ININT IIDNNT --	1	R R D Q					I2 I2 I2 I2	
Absolute value y = |x|	ABS (A)	-- -- IABS I2ABS IIABS IABS JIABS -- ABS DABS QABS CABS	1	I1 I2 I2 I2 I2 I4 I4 I8 R D Q C					I1 I2 I2 I2 I2 I4 I4 I8 R D Q R	Elemental function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
		CDABS CQABS		DC QC					D Q	
Remainder x1-int(x1/x2)*x2 where x2 /= 0 (*8)	MOD (A,P)	-- I2MOD IMOD MOD JMOD MOD -- AMOD DMOD QMOD	2	I1 I2 I2 I2 I4 I4 I8 R D Q	I1 I2 I2 I2 I4 I4 I8 R D Q				I1 I2 I2 I2 I4 I4 I8 R D Q	Elemental function
Change of sign if B >= 0 |A| if B < 0 -|A| (*2)	SIGN (A,B)	-- -- I2SIGN IISIGN ISIGN ISIGN JISIGN -- SIGN DSIGN QSIGN	2	I1 I2 I2 I2 I2 I4 I4 I8 R D Q	I1 I2 I2 I2 I2 I4 I4 I8 R D Q				I1 I2 I2 I2 I2 I4 I4 I8 R D Q	Elemental function
Amount of excess y = x1 - x2 if x1 > x2 , y = 0 if x1 <= x2	DIM (X,Y)	-- -- I2DIM IDIM IIDIM IDIM JIDIM -- DIM DDIM QDIM	2	I1 I2 I2 I2 I2 I4 I4 I8 R D Q	I1 I2 I2 I2 I2 I4 I4 I8 R D Q				I1 I2 I2 I2 I2 I4 I4 I8 R D Q	Elemental function
Double-precision multiplication y=DBLE(x1)*DBLE(x2)	DPROD (X,Y)	DPROD	2	R	R				D	Elemental function
Quadruple-precision multiplication y=QEXTD(x1)* QEXTD(x2)	--	QPROD	2	D	D				Q	Elemental function
Maximum value y=max(x1,x2,...) The value of y is set to the maximum value among x1,x2,...	MAX (A1,A2[,A3,...])	-- MAX MAX I2MAX0 IMAX0 MAX0 JMAX0 MAX0 -- AMAX1 DMAX1	2<=	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D	Elemental function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
		QMAX1 --		Q CH	Q CH	Q CH	Q CH	Q CH	Q CH	
	--	AIMAX0 AMAX0 AJMAX0 AMAX0 IMAX1 JMAX1 MAX1	2<=	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	R R R R I2 I4 I4	
Minimum value y=min(x1,x2,...) The value of y is set to the minimum value among x1,x2,...	MIN (A1,A2[,A3,...])	-- MIN MIN I2MIN0 IMIN0 MIN0 JMIN0 MIN0 -- AMIN1 DMIN1 QMIN1 --	2<=	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D Q CH	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D Q CH	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D Q CH	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D Q CH	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D Q CH	I1 I2 I4 I2 I2 I2 I4 I4 I8 R D Q CH	Elemental function
	--	AIMIN0 AMIN0 AJMIN0 AMIN0 IMIN1 JMIN1 MIN1	2<=	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	I2 I2 I4 I4 R R R	R R R R I2 I4 I4	
Imaginary part	AIMAG (Z)	IMAG AIMAG DIMAG QIMAG	1	C C DC QC					R R D Q	Elemental function
Conjugate complex number y=CMPLX (REAL(x),-IMAG(x))	CONJG (Z)	CONJG DCONJG QCONJG	1	C DC QC					C DC QC	Elemental function
Modulo function	MODULO (A,P)	-- -- -- -- -- -- --	2	I1 I2 I4 I8 R D Q	I1 I2 I4 I8 R D Q				I1 I2 I4 I8 R D Q	Elemental function
Least INTEGER greater than or equal to number	CEILING (A[,KIND])	-- -- --	1	R D Q					I4 I4 I4	Elemental function
		-- -- --	2	R D Q	i i i				i (*1) i (*1) i (*1)	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Greatest INTEGER less than or equal to number	FLOOR (A[,KIND])	-- -- --	1	R D Q					I4 I4 I4	Elemental function
		-- -- --	2	R D Q	i i i				i (*1) i (*1) i (*1)	
Exponent part	EXPONENT (X)	--	1	r					I4	Elemental function
Fraction part	FRACTION (X)	-- -- --	1	R D Q					R D Q	Elemental function
Nearest different number	NEAREST (X,S)	-- -- --	2	R D Q	r r r				R D Q	Elemental function
Reciprocal of relative spacing	RRSPACING (X)	-- -- --	1	R D Q					R D Q	Elemental function
Absolute spacing	SPACING (X)	-- -- --	1	R D Q					R D Q	Elemental function
Scaling $y=x1*2^{(x2)}$	SCALE (X,I)	-- -- --	2	R D Q	i i i				R D Q	Elemental function
Set exponent part	SET_EXPONENT (X,I)	-- -- --	2	R D Q	i i i				R D Q	Elemental function
Merge	MERGE (TSOURCE, FSOURCE, MASK)	--	3	ay	a1	l			a1	Elemental function
Conversion of LOGICAL types	LOGICAL (L[,KIND])	--	1	l					L4	Elemental function
		--	2	l	i				l (*1)	
Test for end-of-file value	IS_IOSTAT_END(I)	--	1	i					L4	Elemental function
Test for end-of-record value	IS_IOSTAT_EOR(I)	--	1	i					L4	Elemental function

*1) If the KIND argument is present, the kind type parameter of the result is that specified by KIND.

*2) This processor can distinguish between positive and negative real zero.

*3) If x is of type one-byte, two-byte, four-byte, or eight-byte INTEGER, int(x) is x. If x is of type default REAL, double-precision REAL, or quadruple-precision REAL, int(x) is the largest INTEGER value that does not exceed the absolute value of x, with the sign of x. If x is of type default COMPLEX, double-precision COMPLEX, or quadruple-precision COMPLEX, int(x) is the value obtained by applying the previously mentioned rules to the real part.

*4) If x is of type default REAL, REAL(x) is x. If x is of type INTEGER, double-precision REAL, or quadruple-precision REAL, REAL(x) is x expressed as data of type default REAL. If x is of type default

COMPLEX, double-precision COMPLEX, or quadruple-precision COMPLEX, REAL(x) is the real part of x expressed as data of type default REAL.

*5) If x is of type double-precision REAL, DBLE(x) is x. If x is of type INTEGER, default REAL, or quadruple-precision REAL, DBLE(x) is x expressed as data of type double-precision REAL. If x is of type COMPLEX, double-precision COMPLEX, or quadruple-precision COMPLEX, DBLE(x) is the real part of x expressed as data of type double-precision REAL.

*6) If x is of type default COMPLEX, CMPLX(x) is x. If x, x1, and x2 are of type INTEGER, default REAL, double-precision REAL, or quadruple-precision REAL, CMPLX(x) is a value of type default COMPLEX. The real part is REAL(x) and the imaginary part is zero. CMPLX(x1,x2) is a value of type default COMPLEX with a real part of REAL(x1) and an imaginary part of REAL(x2). If x is of type double-precision COMPLEX or quadruple-precision COMPLEX, CMPLX(x) is a value of type default COMPLEX. The real part is REAL(x) and the imaginary part is REAL(IMAG(x)).

*7) If x is of type double-precision COMPLEX, DCMPLX(x) is x. If x, x1, and x2 are of type INTEGER, default REAL, double-precision REAL, or quadruple-precision REAL, DCMPLX(x) is a value of type double-precision COMPLEX. The real part is DBLE(x) and the imaginary part is zero. DCMPLX(x1,x2) is a value of type double-precision COMPLEX with a real part of DBLE(x1) and an imaginary part of DBLE(x2). If x is of type default COMPLEX or quadruple-precision COMPLEX, DCMPLX(x) is a value of type double-precision COMPLEX. The real part is DBLE(x) and the imaginary part is DBLE(IMAG(x)).

*8) If the result of int(x1/x2) in AMOD, DMOD, and QMOD may not be expressed as a value of type four-byte INTEGER, the result of AMOD, DMOD, and QMOD will be unpredictable.

Note 1: The meanings of notation are as follows:
-- : Denotes that the corresponding generic or specific name does not exist.
I1 : One-byte INTEGER type.
I2 : Two-byte INTEGER type.
I4 : Four-byte INTEGER type.
I8 : Eight-byte INTEGER type.
i : One-byte INTEGER, two-byte INTEGER, four-byte INTEGER, or eight-byte INTEGER type.
R : Default REAL type.
D : Double-precision REAL type.
Q : Quadruple-precision REAL type.
r : Default REAL, double-precision REAL, or quadruple-precision REAL type.
C : Default COMPLEX type.
DC : Double-precision COMPLEX type.
QC : Quadruple-precision COMPLEX type.
l : One-byte LOGICAL, two-byte LOGICAL, four-byte LOGICAL or eight-byte LOGICAL type.
a1 : The same type as the type of the first argument.
ir : One-byte INTEGER, two-byte INTEGER, four-byte INTEGER, eight-byte INTEGER, default REAL, double-precision REAL, or quadruple-precision REAL type.
ay : Any type.
nd : Binary, octal, or hexadecimal constant.
in : One-byte INTEGER, two-byte INTEGER, four-byte INTEGER, eight-byte INTEGER, default REAL, double-precision REAL, quadruple-precision REAL type or binary, octal, or hexadecimal constant.
CH : Default CHARACTER type.

Note 2: Here, x, x1, and x2 represent actual arguments, and xj denotes the j-th actual argument. xr represents the real part of the actual argument x and xi represents the imaginary part of the actual argument x.

Note 3: y represents the result of the function.

Note 4: When intrinsic functions that have the following specific names are used as an actual argument of a procedure, an actual argument in a reference to the corresponding dummy argument (dummy procedure) must be of type four-byte INTEGER.
IABS, MOD, ISIGN, IDIM, FLOAT, DFLOAT, QFLOAT

Note 5: Intrinsic functions which have the following specific names may not be used as an actual argument of a procedure.

MAX, I2MAX0, IMAX0, MAX0, JMAX0, AMAX1, DMAX1, QMAX1, AIMAX0, AMAX0, AJMAX0, IMAX1, MAX1, JMAX1, MIN, I2MIN0, IMIN0, MIN0, JMIN0, AMIN1, DMIN1, QMIN1, AIMIN0, AMIN0, AJMIN0, IMIN1, MIN1, JMIN1

Table A.2 Mathematical Intrinsic Functions

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Square root if REAL x >= 0 $y=x^{\frac{1}{2}}$ if COMPLEX $y=x^{\frac{1}{2}}$ $-\pi$ <arg(x) <= π	SQRT (X)	SQRT DSQRT QSQRT CSQRT CDSQRT CQSQRT	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Cube root $y=x^{\frac{1}{3}}$	CBRT (X)	CBRT DCBRT QCBRT	1	R D Q					R D Q	Elemental function
Exponent y = e^x x < 88.722 x < 709.782 x < 11356.5 xr < 88.722 |xi| < 8.23e5 xr < 709.782 |xi| < 3.53e15 xr < 11356.5 |xi| < 2^{62} * π	EXP (X)	EXP DEXP QEXP CEXP CDEXP CQEXP	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Exponent y = 2^x x < 128.0 x < 1024.0 x < 16384.0	EXP2 (X)	EXP2 DEXP2 QEXP2	1	R D Q					R D Q	
Exponent y = 10^x x < 38.531 x < 308.254 x < 4932.0625	EXP10 (X)	EXP10 DEXP10 QEXP10	1	R D Q					R D Q	
Logarithm y=log(x) x > 0 y=PVlog(x) PV:Principal value x /= 0 + 0 i	LOG (X)	LOG ALOG DLOG QLOG CLOG CDLOG CQLOG	1	R R D Q C DC QC					R R D Q C DC QC	Elemental function
Logarithm y=log10(x) x > 0	LOG10 (X)	LOG10 ALOG10 DLOG10 QLOG10	1	R R D Q					R R D Q	
Logarithm y=log2(x) x > 0	LOG2 (X)	LOG2 ALOG2 DLOG2 QLOG2	1	R R D Q					R R D Q	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Sine y = sin(x) |x| < 8.23e5 |x| < 3.53e15 |x| < 2^{62} * π |xr| < 8.23e5 |xi| < 89.415 |xr| < 3.53e15 |xi| < 710.475 |xr| < 2^{62} * π |xi| < 11357.125	SIN(X) (Argument in radians)	 SIN DSIN QSIN CSIN CDSIN CQSIN	1	 R D Q C DC QC					 R D Q C DC QC	Elemental function
Sine y = sind(x) = sin(π /180 * x) |x| < 4.72e7 |x| < 2.03e17 |x| < 2^{62} * 180	SIND (X) (Argument in degrees)	 SIND DSIND QSIND	1	 R D Q					 R D Q	
Sine y = sinq(x) = sin(π /2 * x) |xr| < 5.24e5 |xi| < 56.92 |xr| < 2.25e15 |xi| < 452.30 |xr| < 2^{63} |xi| < 7230.125	SINQ (X) (Argument in quadrants)	 SINQ DSINQ QSINQ CSINQ CDSINQ CQSINQ	1	 R D Q C DC QC					 R D Q C DC QC	
Cosine y = cos(x) |x| < 8.23e5 |x| < 3.53e15 |x| < 2^{62} * π |xr| < 8.23e5 |xi| < 89.415 |xr| < 3.53e15 |xi| < 710.475 |xr| < 2^{62} * π |xi| < 11357.125	COS(X) (Argument in radians)	 COS DCOS QCOS CCOS CDCOS CQCOS	1	 R D Q C DC QC					 R D Q C DC QC	Elemental function
Cosine y = cosd(x) = cos(π /180 * x) |x| < 4.72e7 |x| < 2.03e17 |x| < 2^{62} * 180	COSD (X) (Argument in degrees)	 COSD DCOSD QCOSD	1	 R D Q					 R D Q	
Cosine y = cosq(x) = cos(π /2 * x) |xr| < 5.24e5 |xi| < 56.92	COSQ (X) (Argument in quadrants)	 COSQ DCOSQ QCOSQ CCOSQ	1	 R D Q C					 R D Q C	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
|xr| < 2.25e15 |xi| < 452.30 |xr| < 2^{63} |xi| < 723012		CDCOSQ CQCOSQ		DC QC					DC QC	
Tangent y = tan(x) |x| < 8.23e5 |x| < 3.53e15 |x| < 2^{62} * π	TAN(X) (Argument in radians)	 TAN DTAN QTAN -- -- --	1	 R D Q C DC QC					 R D Q C DC QC	Elemental function
Tangent y = tand(x) = tan(π /180 * x) |x| < 4.72e7 |x| < 2.03e17 |x| < 2^{63}*90	TAND (X) (Argument in degrees)	 TAND DTAND QTAND	1	 R D Q					 R D Q	
Tangent y = tanq(x) = tan(π /2 * x)	TANQ (X) (Argument in quadrants)	TANQ DTANQ QTANQ	1	R D Q					R D Q	
Cotangent y = cot(x) |x| < 8.23e5 |x| < 3.53e15 |x| < 2^{62} * π	COTAN (X) (Argument in radians)	 COTAN DCOTAN QCOTAN	1	 R D Q					 R D Q	Elemental function
Cotangent y = cotd(x) = cot(π /180 * x) |x| < 4.72e7 |x| < 2.03e17 |x| < 2^{63}*90	COTAND (X) (Argument in degrees)	 COTAND DCOTAND QCOTAND	1	 R D Q					 R D Q	
Cotangent y = cotq(x) = cot(π /2 * x)	COTANQ (X) (Argument in quadrants)	COTANQ DCOTANQ QCOTANQ	1	R D Q					R D Q	
Arc sine y = arcsin(x) |x| <= 1	ASIN(X) (Function value in radians)	ASIN DASIN QASIN -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Arc sine y = arcsind(x) = 180/ π *arcsin(x) |x| <= 1	ASIND (X) (Function value in degrees)	ASIND DASIND QASIND	1	R D Q					R D Q	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Arc sine y = arcsinq(x) = 2/ π *arcsin(x) |x| <= 1	ASINQ (X) (Function value in quadrants)	ASINQ DASINQ QASINQ	1	R D Q					R D Q	
Arc cosine y = arccos(x) |x| <= 1	ACOS (X) (Function value in radians)	ACOS DACOS QACOS -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Arc cosine y = arccosd(x) =180/ π * arccos(x) |x| <= 1	ACOSD (X) (Function value in degrees)	ACOSD DACOSD QACOSD	1	R D Q					R D Q	
Arc cosine y = arccosq(x) = 2/ π * arccos(x) |x| <= 1	ACOSQ (X) (Function value in quadrants)	ACOSQ DACOSQ QACOSQ	1	R D Q					R D Q	
Arc tangent y = arctan(x)	ATAN (X) (Function value in radians)	ATAN DATAN QATAN -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Arc tangent y = arctan(x1/x2) x1 /= 0, x2 /= 0	ATAN (Y,X) (Function value in radians)	-- -- --	2	R D Q	R D Q				R D Q	
Arc tangent y = arctan(x1/x2) x1 /= 0, x2 /= 0	ATAN2 (Y,X) (Function value in radians)	ATAN2 DATAN2 QATAN2	2	R D Q	R D Q				R D Q	
Arc tangent y = 180/ π * arctan(x)	ATAND (Y) (Function value in degrees)	ATAND DATAND QATAND	1	R D Q					R D Q	
Arc tangent y = 180/ π * arctan(x1/x2) x1 /= 0, x2 /= 0	ATAN2D (Y,X) (Function value in degrees)	ATAN2D DATAN2D QATAN2D	2	R D Q	R D Q				R D Q	
Arc tangent y = 2/ π * arctan(x)	ATANQ (X) (Function value in quadrants)	ATANQ DATANQ QATANQ	1	R D Q					R D Q	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Arc tangent y = 2/ π * arctan(x1/x2) x1 /= 0, x2 /= 0	ATAN2Q (Y,X) (Function value in quadrants)	ATAN2Q DATAN2Q QATAN2Q	2	R D Q	R D Q				R D Q	
Hyperbolic sine y = sinh(x) |x| < 89.415 |x| < 710.475 |x| < 11357.125	SINH (X)	SINH DSINH QSINH -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Hyperbolic cosine y = cosh(x) |x| < 89.415 |x| < 710.475 |x| < 11357.125	COSH (X)	COSH DCOSH QCOSH -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Hyperbolic tangent y = tanh(x)	TANH (X)	TANH DTANH QTANH -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Inverse hyperbolic sine y = arcsinh(x)	ASINH (X) (Function value in radians)	-- -- -- -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Inverse hyperbolic cosine y = arccosh(x) x >= 1	ACOSH (X) (Function value in radians)	-- -- -- -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Inverse hyperbolic tangent y=arctanh(x) |x| <= 1.0	ATANH (X)	-- -- -- -- -- --	1	R D Q C DC QC					R D Q C DC QC	Elemental function
Error function $y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$	ERF (X)	ERF DERF QERF	1	R D Q					R D Q	Elemental function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Complementary error function $y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt$	ERFC (X)	ERFC DERFC QERFC	1	R D Q					R D Q	Elemental function
Gamma function $y=\int_0^\infty e^{-t}t^{x-1}dt$ 0 < x < 35.039860 0 < x < 171.6243 0 < x < 1755.0	GAMMA (X)	 GAMMA DGAMMA QGAMMA	1	 R D Q					 R D Q	Elemental function
Log gamma function $y=\log\lvert\int_0^\infty e^{-t}t^{x-1}dt\rvert$ 0 < x < 0.403711e37 0 < x < 2.55634e305 0 < x < 1.048e4928	LGAMMA (X)	 LGAMMA ALGAMA DLGAMA QLGAMA	1	 R R D Q					 R R D Q	Elemental function
Euclidean distance $\sqrt{(x^2+y^2)}$	HYPOT (X,Y)		2	R D Q	R D Q				R D Q	Elemental function

```
Note 1: The meaning of notation used in the table is as follows:
-- : Denotes that the corresponding generic or specific name does not exist.
X  : Denotes that the procedure is elemental or intrinsic.
O  : Denotes that the procedure is not elemental or user-defined.
R  : Default REAL type.
D  : Double-precision REAL type.
Q  : Quadruple-precision REAL type.
C  : Default COMPLEX type.
DC : Double-precision COMPLEX type.
QC : Quadruple-precision COMPLEX type.

Note 2: Here, x, x1, and x2 represent actual arguments, and xj denotes the j-th actual argument. xr
represents the real part of the actual argument x and xi represents the imaginary part of the actual
argument x.

Note 3: y represents the result of the function.
```

Table A.3 Character Intrinsic Functions

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
The collating sequence of character x	ICHAR (C[,KIND])	ICHAR	1	ch1					I4	Elemental function
			2	ch1	i				i (*1)	
The character with a collating sequence of x	CHAR (I[,KIND])	-- CHAR CHAR --	1 or 2	I1 I2 I4 I8	i i i i				ch1 (*1) ch1 (*1) ch1 (*1) ch1 (*1)	
Position of substring (*2)	INDEX (STRING,	INDEX	2	ch	a1				I4	Elemental function
			3	ch	a1	l			I4	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
	SUBSTRING [,BACK,KIND])			ch	a1		i		i (*1)	
			4	ch	a1	l	i		i (*1)	
Comparison of character string by ASCII code (*3)	LGE LGT LLE LLT (STRING_A, STRING_B)	LGE LGT LLE LLT	2	CH1 CH1 CH1 CH1	CH1 CH1 CH1 CH1				L4 L4 L4 L4	Elemental function
ASCII character	ACHAR (I[,KIND])	--	1	i					CH1	Elemental function
			2	i	i				ch1 (*1)	
Position of ASCII character	IACHAR (C[,KIND])	--	1	ch1					I4	Elemental function
			2	ch1	i				i (*1)	
Length without trailing blank	LEN_TRIM (STRING[,KIND])	--	1	ch					I4	Elemental function
			2	ch	i				i (*1)	
Adjust left	ADJUSTL (STRING)	--	1	ch					a1	Elemental function
Adjust right	ADJUSTR (STRING)	--	1	ch					a1	Elemental function
Scan a string for a character in a set	SCAN (STRING, SET [,BACK,KIND])	--	2	ch	a1				I4	Elemental function
			3	ch	a1	l			I4	
				ch	a1		i		i (*1)	
			4	ch	a1	l	i		i (*1)	
Verify the set of characters in a string	VERIFY (STRING, SET [,BACK,KIND])	--	2	ch	a1				I4	Elemental function
			3	ch	a1	l			I4	
				ch	a1		i		i (*1)	
			4	ch	a1	l	i		i (*1)	

*1) If the optional argument KIND is present, the kind type parameter of the result is that specified by KIND.

*2) If character string x2 exists as a substring of x1, the starting position of the substring is the function value y. If x2 does not exist as a substring of x1, y equals zero.

*3) Comparison of character strings is done according to the ASCII collating sequence. The result value y of LGE is true if x1>=x2 , otherwise false. The result value y of LGT is true if x1>x2, otherwise false. The result value y of LLE is true if x1<=x2, otherwise false. The result value y of LLT is true if x1<x2, otherwise false.

Note 1: The meaning of notation used in the table is as follows:
-- : Denotes that the corresponding generic or specific name does not exist.
X   : Denotes that the procedure is elemental or intrinsic.
O   : Denotes that the procedure is not elemental or user-defined.
I1 : One-byte INTEGER type.
I2 : Two-byte INTEGER type.
I4 : Four-byte INTEGER type.
I8 : Eight-byte INTEGER type.
i   : One-byte, two-byte, four-byte, or eight-byte INTEGER type.
CH : Default CHARACTER type.

```
CH1 : Default CHARACTER type whose length is one.
ch  : CHARACTER type.
ch1 : CHARACTER type whose length is one.
L4  : Four-byte LOGICAL type.
l   : One-byte, two-byte, four-byte, or eight-byte LOGICAL type.
a1  : The same type as the 1st argument.

Note 2: Here, x, x1, and x2 represent actual arguments, and xj denotes the j-th actual argument.

Note 3: y represents the result of the function.

Note 4: When the intrinsic function that has the specific name CHAR is used as an actual argument, an
actual argument in a reference to the corresponding dummy argument (dummy procedure) must be of type
four-byte INTEGER.
```

Table A.4 Bit Manipulation Intrinsic Functions

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Logical negation Logical negation of each bit of x	NOT (I)	-- -- INOT NOT JNOT --	1	I1 I2 I2 I4 I4 I8					I1 I2 I2 I4 I4 I8	Elemental function
Logical product Logical product of x1 and x2	IAND (I,J)	-- -- IIAND IAND JIAND -- -- --	2	I1 I2 I2 I4 I4 I8 i nd	I1 I2 I2 I4 I4 I8 nd i				I1 I2 I2 I4 I4 I8 i i	Elemental function
	AND (I,J)	-- -- AND --	2	I1 I2 I4 I8	I1 I2 I4 I8				I1 I2 I4 I8	
Logical sum Logical sum of x1 and x2	IOR (I,J)	-- -- IIOR IOR JIOR -- -- --	2	I1 I2 I2 I4 I4 I8 i nd	I1 I2 I2 I4 I4 I8 nd i				I1 I2 I2 I4 I4 I8 i i	Elemental function
	OR (I,J)	-- -- OR --	2	I1 I2 I4 I8	I1 I2 I4 I8				I1 I2 I4 I8	
Exclusive logical sum Exclusive logical sum of x1 and x2	IEOR (I,J)	-- -- IIEOR IEOR JIEOR --	2	I1 I2 I2 I4 I4 I8	I1 I2 I2 I4 I4 I8				I1 I2 I2 I4 I4 I8	Elemental function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
		-- --		i nd	nd i				i i	
	XOR (I,J)	-- -- XOR --	2	I1 I2 I4 I8	I1 I2 I4 I8				I1 I2 I4 I8	
Logical shift (*1) |x2| <= BIT_SIZE(x1)	ISHFT (I,SHIFT)	-- -- IISHFT ISHFT JISHFT --	2	I1 I2 I2 I4 I4 I8	i i i i i i				I1 I2 I2 I4 I4 I8	Elemental function
Circular shift of the rightmost bits (*2) |x2| <= BIT_SIZE(x1) 0 <= |x2| <= x3 <= BIT_SIZE(x1) x3 > 0	ISHFTC (I,SHIFT [,SIZE])	-- -- IISHFTC -- JISHFTC --	2	I1 I2 I2 I4 I4 I8	i i i i i i				I1 I2 I2 I4 I4 I8	Elemental function
		-- -- IISHFTC -- JISHFTC --	3	I1 I2 I2 I4 I4 I8	i i i i i i	i i i i i i			I1 I2 I2 I4 I4 I8	
Logical left shift (*3)	LSHIFT (I,SHIFT)	-- -- LSHIFT --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	Elemental function
	SHIFTL (I,SHIFT)	-- -- -- --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	
Arithmetic right shift (*4)	RSHIFT (I,SHIFT)	-- -- RSHIFT --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	Elemental function
	SHIFTA (I,SHIFT)	-- -- -- --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	
Logical right shift (*5)	LRSHFT (I,SHIFT)	-- -- LRSHFT --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	Elemental function
	SHIFTR (I,SHIFT)	-- -- -- --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Combined left shift	DSHIFTL (I,J,SHIFT)	-- -- -- -- -- -- -- -- -- -- -- --	3	I1 I2 I4 I8 I1 I2 I4 I8 nd nd nd nd	I1 I2 I4 I8 nd nd nd nd I1 I2 I4 I8	i i i i i i i i i i i i			I1 I2 I4 I8 I1 I2 I4 I8 I1 I2 I4 I8	Elemental function
Combined right shift	DSHIFTR (I,J,SHIFT)	-- -- -- -- -- -- -- -- -- -- -- --	3	I1 I2 I4 I8 I1 I2 I4 I8 nd nd nd nd	I1 I2 I4 I8 nd nd nd nd I1 I2 I4 I8	i i i i i i i i i i i i			I1 I2 I4 I8 I1 I2 I4 I8 I1 I2 I4 I8	Elemental function
Arithmetic shift	ISHA (I,SHIFT)	-- -- -- --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	Elemental function
Circulate shift	ISHC (I,SHIFT)	-- -- -- --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	Elemental function
Logical shift	ISHL (I,SHIFT)	-- -- -- --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	Elemental function
Bit reverse	IBCHNG (I,POS)	-- -- -- --	2	I1 I2 I4 I8	i i i i				I1 I2 I4 I8	Elemental function
Bit set (*6) 0 <=x2 < BIT_SIZE(x1)	IBSET (I,POS)	-- -- IIBSET IBSET JIBSET --	2	I1 I2 I2 I4 I4 I8	i i i i i i				I1 I2 I2 I4 I4 I8	Elemental function
Bit clear (*7) 0 <= x2 < BIT_SIZE(x1)	IBCLR (I,POS)	-- -- IIBCLR IBCLR JIBCLR --	2	I1 I2 I2 I4 I4 I8	i i i i i i				I1 I2 I2 I4 I4 I8	Elemental function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Bit test (*8) 0 <= x2 < BIT_SIZE(x1)	BTEST (I,POS)	-- -- BITEST BJTEST BTEST --	2	I1 I2 I2 I4 I4 I8	i i i i i i				L4 L4 L4 L4 L4 L4	Elemental function
Extract a sequence of bits 0 <= x2+x3 <= BIT_SIZE(x1) x2 > 0 , x3 > 0 (*9)	IBITS (I,POS,LEN)	-- -- IIBITS -- JIBITS --	3	I1 I2 I2 I4 I4 I8	i i i i i i	i i i i i i			I1 I2 I2 I4 I4 I8	Elemental function
Zero extension	IZEXT (A)	--	1	L1					I2	Elemental function
	IZEXT2 (A)	--	1	I2					I2	
	JZEXT (A)	--	1	L4					I4	
	JZEXT2 (A)	--	1	I2					I4	
	JZEXT4 (A)	--	1	I4					I4	
Bit-wise comparison i <= j	BLE (I,J)	-- -- -- --	2	i i nd nd	i nd i nd				L4 L4 L4 L4	Elemental function
Bit-wise comparison i < j	BLT (I,J)	-- -- -- --	2	i i nd nd	i nd i nd				L4 L4 L4 L4	Elemental function
Bit-wise comparison i >= j	BGE (I,J)	-- -- -- --	2	i i nd nd	i nd i nd				L4 L4 L4 L4	Elemental function
Bit-wise comparison i > j	BGT (I,J)	-- -- -- --	2	i i nd nd	i nd i nd				L4 L4 L4 L4	Elemental function
Leftmost I bits set to 1	MASKL (I[,KIND])	-- --	1 2	i i	 i				I4 i (*10)	Elemental function
Rightmost I bits set to 1	MASKR (I[,KIND])	-- --	1 2	i i	 i				I4 i (*10)	Elemental function
Number of leading zero bits	LEADZ (I)	--	1	i					I4	Elemental function
Number of trailing zero bits	TRAILZ (I)	--	1	i					I4	Elemental function
Number of 1 bits	POPCNT (I)	--	1	i					I4	Elemental function
Parity expressed as 0 or 1	POPPAR (I)	--	1	i					I4	Elemental function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Merge of bits under mask	MERGE_BITS (I,J,MASK)	--	3	I1	I1	I1			I1	Elemental function
		--		I2	I2	I2			I2	
		--		I4	I4	I4			I4	
		--		I8	I8	I8			I8	
		--		I1	nd	I1			I1	
		--		I2	nd	I2			I2	
		--		I4	nd	I4			I4	
		--		I8	nd	I8			I8	
		--		nd	I1	I1			I1	
		--		nd	I2	I2			I2	
		--		nd	I4	I4			I4	
		--		nd	I8	I8			I8	
		--		I1	I1	nd			I1	
		--		I2	I2	nd			I2	
		--		I4	I4	nd			I4	
		--		I8	I8	nd			I8	
		--		i	nd	nd			i	
		--		nd	i	nd			i	

*1) If x2<0 , x1 is logically shifted to the right by |x2| bits. If x2>0 , x1 is logically shifted to the left by |x2| bits. If x2=0, no shift is made.

*2) When the number of arguments is two, if x2<0 , x1 is circularly shifted to the right by |x2| bits. If x2>0 , x1 is circularly shifted to the left by |x2| bits. When the number of arguments is three, if x2<0 , the rightmost |x3| bits of x1 are circularly shifted to the right by |x2| bits. If x2>0 , the rightmost |x3| bits of x1 are circularly shifted to the left by |x2| bits. If x2=0, no shift is made regardless of the number of arguments.

*3) x1 is logically shifted to the left by x2 bits.

*4) x1 is arithmetically shifted to the right by x2 bits.

*5) x1 is logically shifted to the right by x2 bits.

*6) Bit x2 for x1 is set to 1. In determining bit x2, the bits are counted from the right, with the rightmost (least significant) bit counted as bit 0.

*7) Bit x2 for x1 is set to 0. In determining bit x2, the bits are counted from the right, with the rightmost (least significant) bit counted as bit 0.

*8) Bit x2 for x1 is tested. If the bit is a 1, the value is "true". If the bit is a 0, the value is "false". In determining bit x2, the bits are counted from the right, with the rightmost (least significant) bit counted as bit 0.

*9) The left x3 bits from the x2-th bit position are extracted and right-adjusted while other bits are set to 0. In determining bit x2, the bits are counted from the right, with the rightmost (least significant) bit counted as bit 0.

*10) If the optional argument KIND is present, the kind type parameter of the result is that specified by KIND.

Note 1: The meaning of notation used in the table is as follows:
-- : Denotes that the corresponding generic or specific name does not exist.
I1 : One-byte INTEGER type.
I2 : Two-byte INTEGER type.
I4 : Four-byte INTEGER type.
I8 : Eight-byte INTEGER type.
i : One-byte, two-byte, four-byte, or eight-byte INTEGER type.

```
L4 : Four-byte LOGICAL type.
nd : Binary, octal, or hexadecimal constant.

Note 2: Here, x, x1, and x2 represent actual arguments, and xj denotes the j-th actual argument.

Note 3: y represents the result of the function.

Note 4: When intrinsic functions that have the following specific names are used as an actual argument
of a procedure, actual arguments in a reference to the corresponding dummy argument (dummy procedure)
must be of type four-byte INTEGER.
ISHFT, IBSET, IBCLR, BTEST, LSHIFT, RSHIFT, LRSHFT
```

Table A.5 Bit Copy Intrinsic Subroutine

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Copy a sequence of bits (*1)	MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	-- -- -- --	5	I1 I2 I4 I8	i i i i	i i i i	I1 I2 I4 I8	i i i i	SU SU SU SU	Elemental Subroutine

```
*1) The left x3 bits from the x2-th bit position in x1 are copied to the place beginning with x5-th
bit position in x4. In determining bits x2 and x5, the bits are counted from the right, with the
rightmost (least significant) bit counted as bit 0. x4 may be the same variable as x1. The relations
0 <= x2+x3 <= BIT_SIZE(x1), 0 <= x5+x3 <= BIT_SIZE(x4), x2 >= 0,x3 >= 0, and x5>= 0 must be satisfied.

Note 1: The meaning of notation used in the table is as follows:
-- : Denotes that the corresponding generic or specific name does not exist.
X  : Denotes that the procedure is elemental or intrinsic.
O  : Denotes that the procedure is not elemental or user-defined.
I1 : One-byte INTEGER type.
I2 : Two-byte INTEGER type.
I4 : Four-byte INTEGER type.
I8 : Eight-byte INTEGER type.
i  : One-byte, two-byte, four-byte, or eight-byte INTEGER type.
SU : This is an intrinsic subroutine, not a function.

Note 2: Here, x, x1, and x2 represent actual arguments, and xj denotes the j-th actual argument.
```

Table A.6 Inquiry Intrinsic Functions

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Character length The length of character type data x	LEN (STRING[,KIND])	LEN	1	ch					I4	Inquiry function
			2	ch	i				i (*3)	
Whether an allocatable variable is currently allocated (*1)	ALLOCATED (ALLOCATABLE)	--	1	ay					L4	Inquiry function
Number of bits (*2)	BIT_SIZE (I)	-- -- -- --	1	I1 I2 I4 I8					I1 I2 I4 I8	Inquiry function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Number of significant digits	DIGITS (X)	-- --	1	i r					I4 I4	Inquiry function
Negligible positive model number	EPSILON (X)	-- -- --	1	R D Q					R D Q	Inquiry function
Base of number	RADIX (X)	-- --	1	i r					I4 I4	Inquiry function
Smallest positive number	TINY (X)	-- -- --	1	R D Q					R D Q	Inquiry function
Largest number	HUGE (X)	-- -- -- -- -- -- --	1	I1 I2 I4 I8 R D Q					I1 I2 I4 I8 R D Q	Inquiry function
Maximum exponent	MAXEXPONENT (X)	--	1	r					I4	Inquiry function
Minimum exponent	MINEXPONENT (X)	--	1	r					I4	Inquiry function
Kind type parameter	KIND (X)	--	1	it					I4	Inquiry function
Optional is present or not	PRESENT (A)	--	1	ay					L4	Inquiry function
Decimal precision	PRECISION (X)	-- -- -- -- -- --	1	R D Q C DC QC					I4 I4 I4 I4 I4 I4	Inquiry function
Decimal exponent range	RANGE (X)	-- -- --	1	i r cm					I4 I4 I4	Inquiry function
Association status of the pointer	ASSOCIATED (POINTER [,TARGET])	--	1	ay					L4	Inquiry function
		--	2	ay	ay				L4	
Lower bounds of array	LBOUND (ARRAY [,DIM,KIND])	--	1	ay					I4	Inquiry function
		--	2	ay	i				I4	
		--		ay		i			i (*3)	
		--	3	ay	i	i			i (*3)	
Upper bounds of array	UBOUND (ARRAY [,DIM,KIND])	--	1	ay					I4	Inquiry function
		--	2	ay	i				I4	
		--		ay		i			i (*3)	
		--	3	ay	i	i			i (*3)	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Shape of array	SHAPE (SOURCE [,KIND])	--	1	ay					I4	Inquiry function
		--	2	ay	i				i (*3)	
Total number of elements	SIZE (ARRAY [,DIM,KIND])	--	1	ay					I4	Inquiry function
		--	2	ay	i				i (*3)	
		--		ay		i			i (*3)	
		--	3	ay	i	i			i (*3)	
Number of command arguments	COMMAND_ ARGUMENT_ COUNT ()	--	0	--					I4	Inquiry function
Newline character	NEW_LINE (A)	--	1	ch					ch1	Inquiry function
Number of bytes of the argument	SIZEOF (X)	--	1	ay					I8	Inquiry function
Storage size in bits	STORAGE_SIZE (A[,KIND])	-- --	1 2	ay ay	 i				I4 i (*3)	Inquiry function
Same dynamic type or an extension	EXTENDS_TYPE_O F (A,MOLD)	--	2	TY	TY				L4	Inquiry function
Same dynamic type	SAME_TYPE_AS (A,B)	--	2	TY	TY				L4	Inquiry function
Contiguity of an array	IS_CONTIGUOUS (ARRAY)	--	1	ay					L4	Inquiry function

*1) The function value y is true if the allocatable variable ALLOCATABLE is currently allocated; otherwise, it is false. The variable ALLOCATABLE must be an allocatable variable with the ALLOCATABLE attribute.

*2) The function value y is the number of bits of x.

*3) If the optional argument KIND is present, the kind type parameter of the result is that specified by KIND.

Note 1: The meaning of notation used in the table is as follows:
-- :Denotes that the corresponding generic or specific name does not exist.
I1 : One-byte INTEGER type.
I2 : Two-byte INTEGER type.
I4 : Four-byte INTEGER type.
I8 : Eight-byte INTEGER type.
i : One-byte, two-byte, four-byte, or eight-byte INTEGER type.
R : Default REAL type.
D : Double-precision REAL type.
Q : Quadruple-precision REAL type.
r : Default REAL, double-precision REAL, or quadruple-precision REAL type.
C : Default COMPLEX type.
DC : Double-precision COMPLEX type.
QC : Quadruple-precision COMPLEX type.
cm : Default COMPLEX, double-precision COMPLEX, or quadruple-precision COMPLEX type.
CH : Default CHARACTER type.
CH1 : Default CHARACTER type whose length is one.
ch : CHARACTER type.
ch1 : CHARACTER type whose length is one.
L4 : Four-byte LOGICAL type.

```
it  : Any type except derived type.
ay  : Any type.
TY  : Derived type.

Note 2: x represents an actual argument.

Note 3: y represents the result of the function.
```

Table A.7 Date and Time Intrinsic Subroutines

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Returns the processor time	CPU_TIME (TIME)	--	1	r					SU	Subroutine
Date on the real-time clock and date in a from compatible with the representations defined in ISO (*1)	DATE_AND_TIME ([DATE] [,TIME] [,ZONE] [,VALUES])	--	1	CH8					SU	Subroutine
		--	2	CH8	CHA				SU	
		--	3	CH8	CHA	CH5			SU	
		--	4	CH8	CHA	CH5	I4		SU	
INTEGER data from a real-time clock (*2)	SYSTEM_CLOCK ([COUNT] [,COUNT_RATE] [,COUNT_MAX])	--	1	I4					SU	Subroutine
		--	2	I4	I4				SU	
		--		I4	r				SU	
		--	3	I4	I4	I4			SU	
		--		I4	r	I4			SU	

```
*1) The leftmost 8 characters of x1 are set to a digit string "ccyymmdd", where cc is the century, yy
is the year, mm is the month, and dd is the day. The leftmost 10 characters of x2 are set to a digit
string "hhmmss.sss", where hh is the hour, mm is the minutes, and ss.sss is the seconds and
milliseconds.

The leftmost 5 characters of x3 are set to a character string "shhmm", where s is the sign, and hh
and mm are the hours and minutes that represent the time difference from Coordinated Universal Time
(UTC).

The 1st element of the array x4 is set to the year, the 2nd element is set to the month, the 3rd
element is set to day, the 4th element is set to the minutes that represents the time difference from
Coordinated Universal Time (UTC), the 5th element is set to the hour in the range of 0 to 23, the 6th
element is set to the minutes in the range of 0 to 59, the 7th element is set to the seconds in the
range of 0 to 60, and the 8th element is set to the milliseconds in the range of 0 to 999.

The actual arguments x1, x2, and x3 must be scalars. The actual argument x4 must be an array whose
size is greater than 8 and rank is one.

*2) The value of the current system time is assigned to x1 in milliseconds. However, if its value (Z)
is over 86399999, the value is the value of MOD(Z,86400000); x2 has the value of 1000 (the value of
the system clock counter per second); and the value of 86399999 (the maximum value of x1) is assigned
into x3. The actual arguments x1, x2, and x3 must be scalars.

Note 1: The meaning of notation used in the table is as follows:
--  : Denotes that the corresponding generic or specific name does not exist.
X   : Denotes that the procedure is elemental or intrinsic.
O   : Denotes that the procedure is not elemental or user-defined.
I4  : Default INTEGER type.
r   : Default REAL, double-precision REAL, or quadruple-precision REAL type.
```

```
CH5: Default CHARACTER type whose length is greater than 5.
CH8: Default CHARACTER type whose length is greater than 8.
CHA: Default CHARACTER type whose length is greater than 10.
SU : This is an intrinsic subroutine, not a function.

Note 2: Here, x1, x2,... represent actual arguments, and xj denotes the j-th actual argument.
```

Table A.8 Address Getting Intrinsic Functions

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Gets address (*1)	LOC (X)	--	1	ay					I8	Utility function

```
*1) The function value is the address of the variable specified by x.

Note 1: The meaning of notation used in the table is as follows:
-- : Denotes that the corresponding generic or specific name does not exist.
I8 : Eight-byte INTEGER type.
ay : Any type.

Note 2: x represents an actual argument.

Note 3: y represents the result value of the function.
```

Table A.9 Coarray Intrinsic Procedures

Function	Generic name (Keyword)	Specific name	number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Atomic definition	ATOMIC_DEFINE(ATOM, VALUE [, STAT])	--	2 or 3	I4(*1)	i	i			SU	Subroutine
				L4(*1)	l	i				
Atomic reference	ATOMIC_REF(VALUE, ATOM [, STAT])	--	2 or 3	i	I4(*1)	i			SU	Subroutine
				l	L4(*1)	i				
Maximum value of each image	CO_MAX(A [, RESULT_IMAGE, STAT, ERRMSG])	--	1 or 2 or 3 or 4	i(*2)	i	i	CH		SU	Subroutine
				r(*2)	i	i	CH			
				ch(*2)	i	i	CH			
Minimum value of each image	CO_MIN(A [, RESULT_IMAGE, STAT, ERRMSG])	--	1 or 2 or 3 or 4	i(*2)	i	i	CH		SU	Subroutine
				r(*2)	i	i	CH			
				ch(*2)	i	i	CH			
Sum of value on each image	CO_SUM(A [, RESULT_IMAGE,	--	1 or 2 or	i(*2)	i	i	CH		SU	Subroutine
				r(*2)	i	i	CH			

Function	Generic name (Keyword)	Specific name	number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
	STAT, ERRMSG])		3 or 4	cm(*2)	i	i	CH			
Transformation from cosubscripts to image index	IMAGE_INDEX(COARRAY, SUB)	--	2	ay(*2)	i				I4	Inquiry function
Lower cobound	LCOBOUND(COARRAY [, DIM, KIND])	--	1	ay(*2)					I4	Inquiry function
			2	ay(*2)	i				I4	
				ay(*2)		i			i(*3)	
			3	ay(*2)	i	i			i(*3)	
The number of images	NUM_IMAGES()	--	0						I4	Transformational function
Cosubscript(s) of invoking image	THIS_IMAGE()	--	0						I4	Transformational function
	THIS_IMAGE(COARRAY [, DIM])		1 or 2	ay(*2)	I4				I4	
Upper cobound	UCOBOUND(COARRAY [, DIM, KIND])	--	1	ay(*2)					I4	Inquiry function
			2	ay(*2)	i				I4	
				ay(*2)		i			i(*3)	
			3	ay(*2)	i	i			i(*3)	

```
-- : Denotes that the corresponding generic or specific name does not exist.
I4 : Four-byte INTEGER type.
i  : One-byte, two-byte, four-byte, or eight-byte INTEGER type.
l  : One-byte, two-byte, four-byte, or eight-byte LOGICAL type.
r  : Default REAL, double-precision REAL, or quadruple-precision REAL type.
cm : Default COMPLEX, double-precision COMPLEX, or quadruple-precision COMPLEX type.
CH : Default CHARACTER type.
ch : CHARACTER type.
L4 : Four-byte LOGICAL type.
ay : Any type.
SU : This is an intrinsic subroutine, not a function.
*1)  It must be a coarray or a coindexed object.
*2)  It must be a coarray.
*3)  If the optional argument KIND is present, the kind type parameter of the result is that
specified by KIND.
```

Table A.10 Other Intrinsic Procedures

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Returns disassociated pointer or unallocated allocatable	NULL ([MOLD])	--	0 or 1	ay					a1 (*3)	Transformational function
Concatenate copies of string	REPEAT (STRING, NCOPIES)	--	2	ch	i				ch	Transformational function

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Remove trailing blank	TRIM (STRING)	--	1	ch					ch	Transformational function
Value of kind parameter of INTEGER	SELECTED_ INT_KIND (R)	--	1	i					I4	Transformational function
Value of kind parameter of REAL (*2)	SELECTED_ REAL_KIND ([P][,R])	--	1 or 2	i	i				I4	Transformational function
Character kind type parameter value,given character set name	SELECTED_ CHAR_KIND (NAME)	--	1	CH1					I4	Transformational function
Treat first argument as if of type of second argument	TRANSFER (SOURCE,MOLD [,SIZE])	--	2	ay	ay				a2	Transformational function
		--	3	ay	ay	i			a2	
Dot product of two rank-one arrays	DOT_PRODUCT (VECTOR_A, VECTOR_B)	--	2	ar	ar				cu	Transformational function
		--	2	l	l				cu	
Matrix multiplication	MATMUL (VECTOR_A, VECTOR_B)	--	2	ar	ar				cu	Transformational function
		--	2	l	l				cu	
Whether all values are true	ALL (MASK [,DIM])	--	1	l					a1	Transformational function
		--	2	l	i				a1	
Whether any value is true	ANY(MASK [,DIM])	--	1	l					a1	Transformational function
		--	2	l	i				a1	
Count the number of true elements	COUNT (MASK [,DIM,KIND])	--	1	l					I4	Transformational function
		--	2	l	i				I4	
		--		l		i			i (*1)	
		--	3	l	i	i			i (*1)	
Maximum value of the elements of array	MAXVAL (ARRAY [,MASK])	-- -- -- -- -- -- -- -- --	1 or 2	I1 I2 I4 I8 R D Q ch	l l l l l l l l				I1 I2 I4 I8 R D Q ch	Transformational function
	MAXVAL (ARRAY, DIM[,MASK])	-- -- -- -- -- -- -- -- --	2 or 3	I1 I2 I4 I8 R D Q ch	i i i i i i i i	l l l l l l l l			I1 I2 I4 I8 R D Q ch	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
Minimum value of the elements of array	MINVAL (ARRAY [,MASK])	-- -- -- -- -- -- -- --	1 or 2	I1 I2 I4 I8 R D Q ch	l l l l l l l l				I1 I2 I4 I8 R D Q ch	Transformational function
	MINVAL (ARRAY, DIM[,MASK])	-- -- -- -- -- -- -- --	2 or 3	I1 I2 I4 I8 R D Q ch	i i i i i i i i	l l l l l l l l			I1 I2 I4 I8 R D Q ch	Transformational function
Product of all elements of array	PRODUCT (ARRAY [,MASK])	-- -- -- -- -- -- -- -- -- --	1 or 2	I1 I2 I4 I8 R D Q C DC QC	l l l l l l l l l l				I1 I2 I4 I8 R D Q C DC QC	Transformational function
	PRODUCT (ARRAY, DIM[,MASK])	-- -- -- -- -- -- -- -- -- --	2 or 3	I1 I2 I4 I8 R D Q C DC QC	i i i i i i i i i i	l l l l l l l l l l			I1 I2 I4 I8 R D Q C DC QC	
Sum of array elements	SUM (ARRAY [,MASK])	-- -- -- -- -- -- -- -- -- --	1 or 2	I1 I2 I4 I8 R D Q C DC QC	l l l l l l l l l l				I1 I2 I4 I8 R D Q C DC QC	Transformational function
	SUM (ARRAY, DIM[,MASK])	-- -- -- -- -- -- --	2 or 3	I1 I2 I4 I8 R D Q	i i i i i i i	l l l l l l l			I1 I2 I4 I8 R D Q	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
		-- -- --		C DC QC	i i i	l l l			C DC QC	
Pack an array into an array of rank one under mask	PACK (ARRAY, MASK[,VECTOR])	--	2	ay	l				a1	Transformational function
		--	3	ay	l	a1			a1	
Reshape array	RESHAPE (SOURCE, SHAPE [,PAD] [,ORDER])	--	2 or 3 or 4	ay	i	a1	i		a1	Transformational function
Replicates array by adding a dimension	SPREAD (SOURCE, DIM,NCOPIES)	--	3	ay	i	i			a1	Transformational function
Unpack an array of rank one into an array under mask	UNPACK (VECTOR, MASK,FIELD)	--	3	ay	l	a1			a1	Transformational function
Circular shift on an array	CSHIFT (ARRAY, SHIFT[,DIM])	--	2 or 3	ay	i	i			a1	Transformational function
End-off shift on an array	EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])	--	2 or 3 or 4	ay	i	a1	i		a1	Transformational function
Transpose of an array of rank 2	TRANSPOSE (MATRIX)	--	1	ay					a1	Transformational function
Location of a maximum value in an array	MAXLOC (ARRAY [,MASK,KIND])	-- -- --	1	i r ch					I4 I4 I4	Transformational function
		-- -- --	2	i r ch	l l l				I4 I4 I4	
		-- -- --		i r ch		i i i			i (*1) i (*1) i (*1)	
		-- -- --	3	i r ch	l l l	i i i			i (*1) i (*1) i (*1)	
	MAXLOC (ARRAY, DIM[,MASK, KIND])	-- -- --	2	i r ch	i i i				I4 I4 I4	
		-- -- --	3	i r ch	i i i	l l l			I4 I4 I4	

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
		-- -- --		i r ch	i i i		i i i		i (*1) i (*1) i (*1)	
		-- -- --	4	i r ch	i i i	l l l	i i i		i (*1) i (*1) i (*1)	
Location of a minimum value in an array	MINLOC (ARRAY [,MASK,KIND])	-- -- --	1	i r ch					I4 I4 I4	Transformational function
		-- -- --	2	i r ch	l l l				I4 I4 I4	
		-- -- --		i r ch		i i i			i (*1) i (*1) i (*1)	
		-- -- --	3	i r ch	l l l	i i i			i (*1) i (*1) i (*1)	
	MINLOC (ARRAY, DIM[,MASK, KIND])	-- -- --	2	i r ch	i i i				I4 I4 I4	
		-- -- --	3	i r ch	i i i	l l l			I4 I4 I4	
		-- -- --		i r ch	i i i		i i i		i (*1) i (*1) i (*1)	
		-- -- --	4	i r ch	i i i	l l l	i i i		i (*1) i (*1) i (*1)	
Pseudorandom number	RANDOM_NUMBER (HARVEST)	--	1	r					SU	Subroutine
Initializes or restarts the pseudorandom number generator	RANDOM_SEED ([SIZE] [,PUT] [,GET])	--	0	--					SU	Subroutine
		--	1	I4					SU	
		--	1		I4				SU	
		--	1			I4			SU	
Entire command	GET_COMMAND ([COMMAND] [,LENGTH] [,STATUS])	--	0 or 1 or 2 or 3	CH	I4	I4			SU	Subroutine
Command argument	GET_COMMAND _ARGUMENT (NUMBER [,VALUE]	--	1 or 2 or 3	I4	CH	I4	I4		SU	Subroutine

Function	Generic name (Keyword)	Specific name	Number of arguments	Type of argument *n*					Type of function	Class
				1	2	3	4	5		
	[,LENGTH] [,STATUS])		or 4							
Value of an environment variable	GET_ENVIRONMENT _VARIABLE (NAME [,VALUE] [,LENGTH] [,STATUS] [,TRIM_NAME])	--	1 or 2 or 3 or 4 or 5	CH	CH	I4	I4	I	SU	Subroutine
Moves an allocation	MOVE_ALLOC (FROM,TO)	--	2	ay	a1				SU	Subroutine
Gives the actual argument as an immediate value	VAL (X)	-- -- -- -- -- -- -- --	1	I1 I2 I4 L1 L2 L4 R D					I1 I2 I4 L1 L2 L4 R D	Utility function
Allocates a block of memory	MALLOC (SIZE)	-- -- -- --	1	I1 I2 I4 I8					I8 I8 I8 I8	Utility function
Frees a block of memory that is currently allocated	FREE (ADDR)	--	1	I8					SU	Subroutine

*1) If the optional argument KIND is present, the kind type parameter of the result is that specified by KIND.

*2) At least one argument must be present.

*3) The result is a pointer with disassociated association status or unallocated allocatable object.

Note 1: The meaning of notation used in the table is as follows:
-- : Denotes that the corresponding generic or specific name does not exist.
I1 : One-byte INTEGER type.
I2 : Two-byte INTEGER type.
I4 : Four-byte INTEGER type.
I8 : Eight-byte INTEGER type.
i : One-byte, two-byte, four-byte, or eight-byte INTEGER type.
R : Default REAL type.
D : Double-precision REAL type.
Q : Quadruple-precision REAL type.
r : Default REAL, double-precision REAL, or quadruple-precision REAL type.
C : Default COMPLEX type.
DC : Double-precision COMPLEX type.
QC : Quadruple-precision COMPLEX type.
CH : Default CHARACTER type.
CH1 : Default CHARACTER type whose length is one.
ch : CHARACTER type.
ch1 : CHARACTER type whose length is one.

L4 : Four-byte LOGICAL type.
l   : One-byte, two-byte, four-byte, or eight-byte LOGICAL type.
a1 : The same type as the 1st argument.
a2 : The same type as the 2nd argument.
cu : The type of the operation result of the 1st argument and the 2nd argument.
ar : One-byte INTEGER, two-byte INTEGER, four-byte INTEGER, eight-byte INTEGER, default REAL, double-precision REAL, quadruple-precision REAL, default COMPLEX, double-precision COMPLEX, and quadruple-precision COMPLEX type.
ay : Any type.
SU : This is an intrinsic subroutine, not a function.

Note 2: x represents an actual argument.

Note 3: y represents the result value of the function.

Appendix B Service Routines

Service routines are functions and subroutines that are offered as external procedures, and a module SERVICE_ROUTINES is offered, too; that includes explicit interface block for all service routines.

A service function can be referenced in a function reference. A service subroutine can be referenced by the service subroutine name in a CALL statement. All service routines are not pure procedures.

The validity of the number and type of arguments is checked if the module SERVICE_ROUTINES that includes all service routines explicit interface block is used. If you use the module SERVICE_ROUTINES in USE statement, specify the ONLY keyword and the service routine name that only using in the program unit, because the module SERVICE_ROUTINES includes all service routines' explicit interface block. The validity of the number and type of arguments is not checked if the module SERVICE_ROUTINES is not used.

For detailed information on individual service routines, see "Chapter 2 Alphabetical Reference".

Table B.1 Service Subroutine

Description	Service Subroutine name (Keyword argument)	Number of argument	Argument
Compulsion abend	ABORT	0	None
Bit clear	BIC (POS, I)	2	POS : default INTEGER scalar I : default INTEGER scalar
Bit set	BIS (POS, I)	2	POS : default INTEGER scalar I : default INTEGER scalar
Gets CPU time	CLOCK (G, I1, I2)	3	G : four-byte INTEGER, eight-byte INTEGER, single precision REAL, double precision REAL, or quadruple precision REAL I1 : default INTEGER scalar I2 : default INTEGER scalar
Gets CPU time	CLOCKM (I)	1	I : four-byte INTEGER
Gets CPU time	CLOCKV (G1, G2, I1, I2)	4	G1 : four-byte INTEGER, eight-byte INTEGER, single precision REAL, double precision REAL, or quadruple precision REAL G2 : four-byte INTEGER, eight-byte INTEGER, single precision REAL, double precision REAL, or quadruple precision REAL I1 : default INTEGER scalar I2 : default INTEGER scalar
Gets date	DATE (CH)	1	CH : default CHARACTER scalar
Tests for a division by zero	DVCHK (I)	1	I : default INTEGER scalar
Outputs the character string to the standard output file	ERROR (STRING)	1	STRING : default CHARACTER scalar
Saves the error item	ERRSAV (ERRNO, DAREA)	2	ERRNO : default INTEGER scalar DAREA : default CHARACTER scalar that length is 16 bytes
Changes the contents of the error control table	ERRSET (ERRNO, ESTOP, MPRINT, TRACE, UEXIT, R)	6	ERRNO : default INTEGER scalar ESTOP : default INTEGER scalar MPRINT : default INTEGER scalar TRACE : default INTEGER scalar UEXIT : default INTEGER scalar R : default INTEGER scalar
Stores the error item	ERRSTR (ERRNO, DAREA)	2	ERRNO : default INTEGER scalar DAREA : default CHARACTER scalar that length is 16 bytes

Description	Service Subroutine name (Keyword argument)	Number of argument	Argument
Outputs the trace back map	ERRTRA	0	None
Terminates the execution	EXIT	0	None
Gets the time and date	FDATE (STRING)	1	STRING : default CHARACTER scalar with length greater than or equal to 24
Outputs data on buffer	FLUSH (UNIT)	1	UNIT : default INTEGER scalar
Free the memory	FREE (ADDR)	1	ADDR : Eight-byte INTEGER scalar
Gets the option character string	GETARG (ARGNO, ARGST)	2	ARGNO : default INTEGER scalar ARGST : default CHARACTER scalar
Gets the argument character string	GETCL (STRING)	1	STRING : default CHARACTER scalar
Gets date	GETDAT (YEAR, MONTH, DAY)	3	YEAR : two-byte INTEGER scalar MONTH : two-byte INTEGER scalar DAY : two-byte INTEGER scalar
Gets the value of environment variable	GETENV (ENV , STRING)	2	ENV : default CHARACTER scalar STRING : default CHARACTER scalar
Gets the login name	GETLOG (NAME)	1	NAME : default CHARACTER scalar
Gets the length of options and the character string	GETPARM (LEN, PARM)	2	LEN : default INTEGER scalar PARM : default CHARACTER scalar
Gets time	GETTIM (HOUR, MINUTES, SECOND, SECOND1_100)	4	HOUR : two-byte INTEGER scalar MINUTES : two-byte INTEGER scalar SECOND : two-byte INTEGER scalar SECOND1_100 : two-byte INTEGER scalar
Returns relative time-of-day	GETTOD (G)	1	G : double precision REAL scalar
Gets the Greenwich mean time	GMTIME (TIME, T)	2	TIME : default INTEGER scalar T : default INTEGER array of rank one
Converts to the character of the absolute value	IBTOD (I, J)	2	I : default INTEGER scalar J : default INTEGER scalar
Gets date	IDATE (IA)	1	IA : default INTEGER of rank one
Converts to IBM370-format floating-point data	IETOM (R1, R2, TYPE, RETCD)	4	R1 : default REAL or double precision REAL scalar R2 : default REAL or double precision REAL scalar TYPE : default INTEGER scalar RETCD : default INTEGER scalar
Gets the diagnostic message at the execution	IOSTAT_MSG (ERRNUM, MESSAGE)	2	ERRNUM : default INTEGER scalar MESSAGE : default CHARACTER scalar
Gets time	ITIME (IA)	1	IA : default INTEGER array of rank one
Transfers the memory	IVALUE (I, J, K)	3	I : default INTEGER scalar J : default INTEGER scalar K : default INTEGER scalar
Gets the local time	LTIME (TIME, T)	2	TIME : default INTEGER scalar T : default INTEGER of rank one
Converts from the IBM370-format floating-point data	MTOIE (R1, R2, TYPE, RETCD)	4	R1 : default REAL or double precision REAL scalar R2 : default REAL or double precision REAL

Description	Service Subroutine name (Keyword argument)	Number of argument	Argument
			scalar TYPE : default INTEGER scalar RETCD : default INTEGER scalar
Tests for an exponent overflow or underflow exception	OVERFL (I)	1	I : default INTEGER scalar
Outputs the character string to the standard error output file	PERROR (STRING)	1	STRING : default CHARACTER scalar
Sets fill character for numeric fields that are wider than supplied numeric precision	PRECFILL (LETTER)	1	LETTER : 1-byte default CHARACTER scalar
Changes the number of precision lowering	PRNSET (I)	1	I : default INTEGER scalar
Sets the prompt message	PROMPT (STRING)	1	STRING : default CHARACTER scalar
Quick sort on an array of rank one	QSORT (ARRAY, NEL, WIDTH, COMPARE)	4	ARRAY : any type of rank one NEL : default INTEGER scalar WIDTH : default INTEGER scalar COMPARE : two-byte INTEGER
Gets the Fortran record length that was read immediately before	REDLEN (I, RETCD)	2	I : default INTEGER scalar RETCD : default INTEGER scalar
Sets bit	SETBIT (POS, I, STATUS)	3	POS : default INTEGER scalar I : default INTEGER scalar STATUS : default INTEGER scalar
Sets the return code	SETRCD (I)	1	I : default INTEGER scalar
Suspends the process	SLEEP (I)	1	I : default INTEGER scalar
Turns sense lights on or off	SLITE (I)	1	I : default INTEGER scalar
Tests sense lights on or off	SLITET (I , J)	2	I : default INTEGER scalar J : default INTEGER scalar
Gets the hundredths of seconds since midnight	TIMER (IX)	1	IX : default INTEGER scalar

Table B.2 Service Function

Description	Service Function name (Keyword argument)	Number of argument	Argument	Type of Return value
Determines if a file exists and how it can be accessed	ACCESS (FNAME , MODE)	2	FNAME : default CHARACTER scalar MODE : default CHARACTER scalar	default INTEGER scalar
Executes a subroutine after specified time	ALARM (TIME, SUB)	2	TIME: default INTEGER scalar SUB: name of the subroutine	default INTEGER scalar
bit test	BIT (POS, I)	2	POS : default INTEGER scalar I : default INTEGER scalar	default LOGICAL scalar

Description	Service Function name (Keyword argument)	Number of argument	Argument	Type of Return value
Changes the default directory	CHDIR (DIRNAME)	1	DIRNAME : default CHARACTER scalar	default INTEGER scalar
Changes the attributes of a file	CHMOD (NAME, MODE)	2	NAME : default CHARACTER scalar MODE : default CHARACTER scalar	default INTEGER scalar
Gets the system time	CTIME (TIME)	1	TIME : default INTEGER scalar	default CHARACTER scalar
Gets the random number	DRAND (I)	1	I : default INTEGER scalar	double precision REAL scalar
Acquisition of CPU time	DTIME (TM)	1	TM : default REAL of rank one	default REAL scalar
Gets CPU time	ETIME (TM)	1	TM : default REAL of rank one	default REAL scalar
Reads a character from the file	FGETC (UNIT, CH)	2	UNIT : default INTEGER scalar CH : 1 byte default CHARACTER	default INTEGER scalar
Creates a copy of the calling process	FORK ( )	0	None	default INTEGER scalar
Writes a character to the file	FPUTC (UNIT, CH)	2	UNIT : default INTEGER scalar CH : 1 byte default CHARACTER	default INTEGER scalar
Positions a file connected	FSEEK (UNIT, OFFSET, FROM)	3	UNIT : default INTEGER scalar OFFSET : default INTEGER scalar FROM : default INTEGER scalar	default INTEGER scalar
Positions a file connected	FSEEKO64 (UNIT, OFFSET, FROM)	3	UNIT : default INTEGER scalar OFFSET : eight-byte INTEGER scalar FROM : default INTEGER scalar	default INTEGER scalar
Gets the information of a file condition	FSTAT (UNIT, STATUS)	2	UNIT : default INTEGER scalar STATUS : default INTEGER of rank one	default INTEGER scalar
Gets the information of a file condition	FSTAT64 (UNIT, STATUS)	2	UNIT : default INTEGER scalar STATUS : eight-byte INTEGER of rank one	default INTEGER scalar
Gets the file position from the file	FTELL (UNIT)	1	UNIT : default INTEGER scalar	default INTEGER scalar
Gets the file position from the file	FTELLO64 (UNIT)	1	UNIT : default INTEGER scalar	eight-byte INTEGER scalar
Reads the next available character from the standard input file	GETC (CH)	1	CH : default CHARACTER scalar	default INTEGER scalar
Gets the current directory pathname	GETCWD (CH)	1	CH : default CHARACTER scalar	default INTEGER scalar

Description	Service Function name (Keyword argument)	Number of argument	Argument	Type of Return value
Gets the file descriptor	GETFD (UNIT)	1	UNIT : default INTEGER scalar	default INTEGER scalar
Gets the group ID	GETGID ( )	0	None	default INTEGER scalar
Gets the process ID	GETPID ( )	0	None	default INTEGER scalar
Gets the user ID	GETUID ( )	0	None	default INTEGER scalar
Gets the host name	HOSTNM (NAME)	1	NAME : default CHARACTER scalar	default INTEGER scalar
Gets the number of the option character string	IARGC ( )	0	None	default INTEGER scalar
Gets of the system error code	IERRNO ( )	0	None	default INTEGER
Gets the maximum number of type integer	INMAX ( )	0	None	default INTEGER scalar
Set the following information of input/ output	IOINIT (CCTL, BLANK, POSITION, PREFIX, INFO)	5	CCTL : default LOGICAL scalar BLANK : default LOGICAL scalar POSITION : default LOGICAL scalar PREFIX : default CHARACTER scalar INFO : default LOGICAL scalar	default LOGICAL scalar
Generates an INTEGER random number	IRAND (I)	1	I : default INTEGER scalar	default INTEGER scalar
Whether a unit is associated with a terminal device	ISATTY (UNIT)	1	UNIT : default INTEGER scalar	default LOGICAL scalar
Gets Julian date	JDATE ( )	0	None	eight-byte default CHARACTER scalar
Kills the process	KILL (PID, SIG)	2	PID : default INTEGER scalar SIG : default INTEGER scalar	default INTEGER scalar
Makes a link to an existing file	LINK (PATH1, PATH2)	2	PATH1 : default CHARACTER scalar PATH2 : default CHARACTER scalar	default INTEGER scalar
Locates the position of the last non blank character	LNBLNK (STRING)	1	STRING : default CHARACTER scalar	default INTEGER scalar

Description	Service Function name (Keyword argument)	Number of argument	Argument	Type of Return value
Converts to of type default INTEGER	LONG (IX)	1	IX : two-byte INTEGER scalar	default INTEGER
gets the information about a file	LSTAT (NAME, STATUS)	2	NAME : default CHARACTER scalar STATUS : default INTEGER of rank one	default INTEGER scalar
gets the information about a file	LSTAT64 (NAME, STATUS)	2	NAME : default CHARACTER scalar STATUS : eight-byte INTEGER of rank one	default INTEGER scalar
Allocates a block of memory	MALLOC (SIZE)	1	SIZE : Eight-byte INTEGER scalar, its same as the result	Eight-byte INTEGER scalar
Gets the number of the option character string	NARGS ( )	0	None	default INTEGER scalar
Writes a character to the standard output file	PUTC (CH)	1	CH : 1 byte default CHARACTER	default INTEGER scalar
Generates a random number	RAN (SEED)	1	SEED : default INTEGER scalar	single precision REAL scalar
Generates a random number	RAND (I)	1	I : default INTEGER scalar	single precision REAL scalar
Renames a file	RENAME (OLD, NEW)	2	OLD : default CHARACTER scalar NEW : default CHARACTER scalar	default INTEGER scalar
Locates the index of the rightmost occurrence of a substring within a string	RINDEX (STRING, SUBSTRING)	2	STRING : default CHARACTER scalar SUBSTRING : default CHARACTER scalar	default INTEGER scalar
Gets the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1,1970	RTC ( )	0	None	double precision REAL scalar
Gets the number of seconds that have elapsed since midnight	SECNDS (SEC)	1	SEC : default REAL scalar	default REAL scalar
Gets the user CPU time	SECOND ( )	0	None	default REAL scalar
Sends a command to the Bourne shell	SH (COMMAND)	1	COMMAND : default CHARACTER scalar	default INTEGER scalar
Converts to a 2-byte integer	SHORT (IX)	1	IX : default INTEGER scalar	two-byte INTEGER scalar
Specifies the action in signal occurrence	SIGNAL (I, FUNC, FLAG)	3	I : default INTEGER scalar FUNC : name of the function FLAG : default INTEGER scalar	default INTEGER scalar

Description	Service Function name (Keyword argument)	Number of argument	Argument	Type of Return value
Gets the information about a file	STAT (NAME, STATUS)	2	NAME : default CHARACTER scalar STATUS : default INTEGER of rank one	default INTEGER scalar
Gets the information about a file	STAT64 (NAME, STATUS)	2	NAME : default CHARACTER scalar STATUS : eight-byte INTEGER of rank one	default INTEGER scalar
Makes a symbolic link to an existing file	SYMLNK (PATH1, PATH2)	2	PATH1 : default CHARACTER scalar PATH2 : default CHARACTER scalar	default INTEGER scalar
Sends a command to the shell	SYSTEM (CH)	1	CH : default CHARACTER scalar	default INTEGER scalar
Gets the system time, in seconds, since 00:00:00 Greenwich mean time, January 1,1970	TIME ( )	0	None	default INTEGER scalar
Gets the time since the first time it is called	TIMEF ( )	0	None	double precision REAL scalar
Gets the path name of the terminal device	TTYNAM (UNIT)	1	UNIT : default INTEGER scalar	default CHARACTER scalar
Removes a specified directory entry	UNLINK (NAME)	1	NAME : default CHARACTER scalar	default INTEGER scalar
suspends a process	WAIT (STATUS)	1	STATUS : default INTEGER scalar	default INTEGER scalar

Appendix C Extended Features

The table below offers extended features from ISO Fortran.

Table C.1 Language Extensions to ISO Fortran

[Fortran character set] - Currency symbol as a letter ($) - Escape sequence \ (backslash)
[Source program] - Fixed source form Unlimited number of continuation lines - Free source form Insignificant blank Unlimited number of continuation lines
[COARRAY specifications] - Coarray Explicit coshape coarray Allocatable coarray - CODIMENSION attribute, CODIMENSION statement - Coarray specifier in ALLOCATABLE statement, TARGET statement, Type declaration statement - Image selector - Execution statements Specify coarray in ALLOCATE or DEALLOCATE statement CRITICAL constructor, END CRITICAL statement, LOCK statement, SYNC ALL statement, SYNC IMAGE statement, SYNC MEMORY statement, UNLOCK statement - Intrinsic procedures ATOMIC_DEFINE, ATOMIC_REF, CO_MAX, CO_MIN, CO_SUM, IMAGE_INDEX, LCOBOUND, NUM_IMAGES, THIS_IMAGE, UCOBOUND Specify coarray in MOVE_ALLOC - ISO_FORTRAN_ENV intrinsic module - LOCK_TYPE type, ATOMIC_INT_KIND, ATOMIC_LOGICAL_KIND, STAT_LOCKED, STAT_LOCKED_OTHER_IMAGE, STAT_STOPPED_IMAGE, STAT_UNLOCKED
[Statements] - Executable statements DO CONCURRENT statement DO UNTIL statement ERROR STOP statement - Nonexecutable statements STRUCTURE statement END STRUCTURE statement UNION statement

END UNION statement

MAP statement

END MAP statement

RECORD statement

POINTER statement (CRAY pointer)

AUTOMATIC statement

STATIC statement

BYTE statement

CHANGEENTRY statement

CONTIGUOUS statement

[Attribute]

- AUTOMATIC attribute
- CHANGEENTRY attribute
- CONTIGUOUS attribute
- STATIC attribute
- Implicit SAVE attribute in declaration part of module

[Data types]

- Derived type definitions by STRUCTURE statement

  STRUCTURE and END STRUCTURE statement
- Specification of union

  UNION, END UNION, MAP, END MAP statement

[Constants]

- Hollerith constants
- Octal constants

  '*digits*'O , "*digits*"O
- Hexadecimal constants

  X'*digits*' , X"*digits*", '*digits*'X , "*digits*"X
- Binary, octal, and hexadecimal constants may be specified as follows.
  - A PARAMETER statement and an initialization in a type declaration statement
  - An argument of intrinsic function BGE, BGT, BLE, BLT, DSHIFTL, DSHIFTR, IAND,IEOR, IOR or MERGE_BITS

[Variables]

- Structure component selection notation (.)

[Type declarations and specifications]

- Type declaration statements

  Kind (size) specified with asterisk

  Initialization (/... /)
- RECORD statement
- BYTE statement

- Implied shape array
- Double colon after PROCEDURE in interface block

[IMPLICIT statement]

- IMPLICIT UNDEFINED
- Kind (size) specified with asterisk
- Currency symbol as a letter

[POINTER statement (CRAY pointer)]

- CRAY pointer statement

[Unformatted input/output statements]

- NUM= specifier

[OPEN statement]

- The value BOTH in ACTION= specifier
- The value BINARY in FORM= specifier
- BLOCKSIZE= specifier
- TOTALREC= specifier
- CARRIAGECONTROL= specifier
- CONVERT= specifier
- The value SHR in STATUS= specifier
- The value APPEND in ACCESS= specifier

[INQUIRE statement]

- BINARY= specifier
- BLOCKSIZE= specifier
- CARRIAGECONTROL= specifier
- CONVERT= specifier
- FLEN= specifier

[CLOSE statement]

- The value FSYNC in STATUS= specifier

[Format specifications]

- Edit descriptors

  G*w* edit descriptor

  $ edit descriptor

  Q*w.d* edit descriptor

  Q remain size edit descriptor

  R edit descriptor

  \ edit descriptor

  D, E, F, G, I, L, B, O, and Z edit descriptor without *w*, *d*, or *e* indicators

- Namelist Formatting

  Namelist record (&*name* ... &end)

[Program units]

- Function subprogram

  Length (size) specified with asterisk

[Intrinsic module function]

- C_SIZEOF
- COMPILER_OPTIONS
- COMPILER_VERSION

[ISO_FORTRAN_ENV Intrinsic Module]

- INT8
- INT16
- INT32
- INT64
- REAL32
- REAL64
- REAL128
- CHARACTER_KINDS
- INTEGER_KINDS
- LOGICAL_KINDS
- REAL_KINDS

[Intrinsic procedures]

- Generic name

  DCMPLX
  CBRT, EXP2, EXP10, LOG2, COTAN, ERF, ERFC, GAMMA, LGAMMA
  SIND, COSD, TAND, ASIND, ACOSD, ATAND, ATAN2D
  SINQ, COSQ, TANQ, COTAND, COTANQ, ASINQ, ACOSQ, ATANQ, ATAN2Q
  AND, OR, XOR, LSHIFT, RSHIFT, LRSHFT
  LOC
  JFIX, INT1, INT2, INT4, I2NINT, IBCHNG, ISHA, ISHC, ISHL, IZEXT, JZEXT,
  IZEXT2, JZEXT2, JZEXT4, VAL
  SIZEOF
  ACOSH, ASINH, ATANH, BGE, BGT, BLE, BLT, DSHIFTL, DSHIFTR, HYPOT,
  LEADZ, MASKL, MASKR, MERGE_BITS, POPCNT, POPPAR, SHIFTA, SHIFTL,
  SHIFTR, STORAGE_SIZE, TRAILZ, IS_CONTIGUOUS

- Specific name

  HFIX
  DFLOAT, DBLE, DREAL, CMPLX, DCMPLX, CDABS, MAX, MIN, IMAG, DIMAG,
  DCONJG
  CDSQRT, CDEXP, CDLOG, CDSIN, SIND, DSIND, COSD, DCOSD, TAND, DTAND,
  ASIND, DASIND, ACOSD, DACOSD, ATAND, DATAND, ATAN2D, DATAN2D
  CBRT, DCBRT, EXP2, DEXP2, EXP10, DEXP10, LOG, LOG10, LOG2, ALOG2,
  DLOG2, SINQ, DSINQ, COSQ, DCOSQ, TANQ, DTANQ, COTAN, DCOTAN,
  COTAND, DCOTAND, COTANQ, DCOTANQ, ASINQ, DASINQ, ACOSQ, DACOSQ,
  ATANQ, DATANQ, ATAN2Q, DATAN2Q, ERF, DERF, ERFC, DERFC, GAMMA,
  DGAMMA, LGAMMA, ALGAMA, DLGAMA
  NOT, IAND, IOR, IEOR, ISHFT, IBSET, IBCLR, BTEST, AND, OR, XOR, LSHIFT,
  RSHIFT, LRSHFT
  AIMAX0, AJMAX0, I2MAX0, IMAX0, JMAX0, IMAX1, JMAX1, AIMIN0, AJMIN0,
  I2MIN0, IMIN0, JMIN0, IMIN1, JMIN1, FLOATI, FLOATJ, DFLOTI, DFLOTJ, IIABS,

JIABS, I2ABS, IIDIM, JIDIM, I2DIM, IIFIX, JIFIX, IINT, JINT, ININT, JNINT, IIDNNT, JIDNNT, IIDINT, JIDINT, IMOD, JMOD, I2MOD, IISIGN, JISIGN, I2SIGN, BITEST, BJTEST, IIBCLR, JIBCLR, IIBITS, JIBITS, IIBSET, JIBSET, IIAND, JIAND, IIEOR, JIEOR, IIOR, JIOR, INOT, JNOT, IISHFT, JISHFT, IISHFTC, JISHFTC

- Argument is of type COMPLEX

  ACOS, ASIN, ATAN, COSH, SINH, TAN, TANH

- Two arguments intrinsic function ATAN

[Constructs]

- BLOCK construct, END BLOCK statement
- Default REAL types and double precision REAL types can be specified in DO variables and scalar INTEGER expressions in DO construct.

[Specific binding]

- Multiple procedures can be specified in specific type binding declarations.

[ALLOCATE statement]

- The allocate shape specifications list can be omitted if a SOURCE= specifier or MOLD= specifier is specified.
- MOLD= specifier

[C_LOC intrinsic module function]

- Inquiries are possible for array addresses from only scalar addresses.

[Pointer assignment]

- The pointer target is simply contiguous without of rank one when a bound remapping list is specified.

[Optional dummy argument]

- The optional dummy argument is nonpointer, nonallocatable, and the corresponding actual argument is an unallocated or disassociated status.

[STOP statement]

- Scalar default character initialization expression, scalar integer initialization expression.

Appendix D Glossary

abstract type

A type that has the ABSTRACT attribute. Objects that are not polymorphic cannot be declared as abstract types. Polymorphic objects cannot be constructed or allocated such that they have a dynamic abstract type.

action statement

A single statement specifying or controlling a computational action.

actual argument

An expression, a variable, a procedure, or an alternate return specifier that is specified in a procedure reference.

allocatable variable

A variable with the ALLOCATABLE attribute. An allocatable variable can be referenced and defined only when it has been allocated. If it is an array, it has a shape only when it has been allocated. An allocatable variable is a named variable or a structure component.

allocatable coarray

Allocatable coarray is coarray with ALLOCATABLE attribute.

argument

An actual argument or a dummy argument.

argument association

The relationship between an actual argument and a dummy argument during the execution of a procedure reference.

argument keyword

A dummy argument name. It may be used in a procedure reference followed by the equals symbol provided the procedure has an explicit interface.

array

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.

array element

One of the scalar data that make up an array that is either named or is a structure component.

array pointer

A pointer to an array.

array section

A subobject that is an array and is not a structure component.

array-valued

Having the property of being an array.

assignment statement

A statement of the form "*variable* = *expression*".

associate name

Construct entity name associated by a selector in a SELECT TYPE construct or an ASSOCIATE construct.

association

A pointer association, a storage sequence association, or an inheritance association.

assumed-shape array

A nonpointer dummy array that takes its shape from the associated actual argument.

assumed-size array

A dummy array whose size is assumed from the associated actual argument. Its last upper bound is specified by an asterisk.

attribute

A property of a data object that may be specified in a type declaration statement.

automatic data object

A data object that is a local entity of a subprogram, that is not a dummy argument, and that has a nonconstant CHARACTER length or array bound.

belong

If an EXIT or a CYCLE statement contains a construct name, the statement belongs to the DO construct using that name. Otherwise, it belongs to the innermost DO construct in which it appears.

binding label

A default character type value that uniquely identifies a variable, common block, subroutine, or function in a companion processor.

block

A sequence of executable constructs embedded in another executable construct, bounded by statements that are particular to the construct, and treated as an integral unit.

block data program unit

A program unit that provides initial values for data objects in named common blocks.

bounds

For a named array, the limits within which the values of the subscripts of its array elements shall lie.

character

A letter, digit, or other symbol.

characteristics

Characteristics are of a procedure, of a dummy argument, of a dummy data object, of a dummy procedure and of a function result.

character length

Length of a character string.

character length parameter

The type parameter that specifies the number of characters for an entity of type character.

character string

A sequence of characters numbered from left to right 1, 2, 3, . . .

character storage unit

The unit of storage for holding a scalar that is not a pointer and it of type default CHARACTER and character length one.

class

Being an extended type.

coarray

Coarray is a data entity that can be directly referenced or defined by any image for COARRAY specification.

collating sequence

An ordering of all the different characters of a particular kind type parameter.

common block

A block of physical storage that may be accessed by any of the scoping units in a program.

companion processor

A mechanism that can reference, ascertain, and define global data and procedures. It can reference, ascertain, and define these types of entities using means other than Fortran.

component

A constituent of a derived type.

component order

The order of components in a derived type used in intrinsic formatted input/output and structure constructors.

conformable

Two arrays are said to be conformable if they have the same shape. A scalar is conformable with any array.

conformance

A program conforms to the standard if it uses only those forms and relationships described therein and if the program has an interpretation according to the standard. A program unit conforms to the standard if it can be included in a program in a manner that allows the program to be standard conforming.

connected

For an external unit, the property of referring to an external file.

For an external file, the property of having an external unit that refers to it.

constant

A data object whose value shall not change during execution of a program. It may be a named constant or a literal constant.

constant expression

An expression satisfying rules that ensure that its value does not vary during program execution.

construct

A sequence of statements starting with an ASSOCIATE, SELECT CASE, DO, IF, FORALL, SELECT TYPE, or WHERE statement and ending with the corresponding terminal statement.

construct association

The association between a selector in an ASSOCIATE construct or a SELECT TYPE construct and the associate name.

construct entity

An entity identified by a lexical token that sets one structure construct as the scope.

control mask

A logic type array that uses a logic value to determine which array element is defined in each WHERE assignment statement in a WHERE statement or a WHERE construct.

corank

A corank is the number of codimensions of a coarray.

coindexed object

A coindexed object is data object with an image selector.

cosubscript

A cosubscript is scalar integer expression in an image selector. The value of a cosubscript is within the cobounds.

data

Plural of datum.

data entity

A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (called the function result). A data entity has a data type (either intrinsic or derived) and has, or may have, a data value (the exception is an undefined variable). Every data entity has a rank and is thus either a scalar or an array.

data object

A data entity that is a constant, a variable, or a subobject of a constant.

data type

A named category of data that is characterized by a set of values, together with a way to denote these values and a collection of operations that interpret and manipulate the values. For an intrinsic type, the set of data values depends on the values of the type parameters.

datum

A set that has any type and any value.

decimal symbol

The character that separates the integer part and the fractional digits in the decimal representation of real numbers in a file. The implicit decimal symbol is "." (period). The current decimal point edit mode determines the current decimal symbol.

declaration expression

An expression that limits the use of specifications such as length type parameters and array shapes.

declared type

The type of a data entity at the time it is declared. For a polymorphic data entity, this type can be different to the type at execution (the dynamic type).

default initialization

If initialization is specified in a type definition, an object of the type will be automatically initialized. Nonpointer components may be initialized with values by default; pointer components may be initially disassociated by default. Default initialization is not provided for objects of intrinsic type.

deferred binding

A binding that has the DEFERRED attribute. Deferred bindings can only be written in abstract type definitions.

deferred type parameter

A length type parameter that does not specify a value when an object is declared, but specifies a value when that object is allocated or a pointer is assigned.

definable

A variable is definable if its value may be changed by the appearance of its name or designator on the left of an assignment statement. An allocatable array that has not been allocated is an example of a data object that is not definable. An example of a subobject that is not definable is C(I) when C is an array that is a constant and I is an INTEGER variable.

defined

For a data object, the property of having or being given a valid value.

defined assignment statement

An assignment statement that is not an intrinsic assignment statement and is defined by a subroutine and an interface block that specifies ASSIGNMENT (=).

defined operation

An operation that is not an intrinsic operation and is defined by a function that is associated with a generic identifier.

deleted feature

A feature in a previous Fortran standard that is considered to have been redundant and largely unused.

derived type

A data type that has components. The components are intrinsic types or other derived types.

disassociated

A disassociated pointer is not associated with any target. A pointer is disassociated following execution of a DEALLOCATE or NULLIFY statement, or following pointer association with a disassociated pointer.

dummy argument

An entity whose name appears in the parenthesized list following the procedure name in a FUNCTION statement, a SUBROUTINE statement, an ENTRY statement, or a statement function statement.

dummy array

A dummy argument that is an array.

dummy data object

A dummy argument that is a data object.

dummy pointer

A dummy argument that is a pointer.

dummy procedure

A dummy argument that is specified or referenced as a procedure.

dynamic type

The type a data entity has at the time of execution. The dynamic type of a data entity that is not polymorphic is the same as the declared type.

effective item

A scalar object obtained by expanding an input/output list.

elemental

An adjective applied to an intrinsic operation, procedure, or assignment statement that is applied independently to elements of an array or corresponding elements of a set of conformable arrays and scalars.

entity

The term used for any of the following: a program unit, a procedure, an abstract interface, an operator, a generic interface, a common block, an external unit, a statement function, a type, a data entity, a statement label, label, construct, or a namelist group.

executable construct

An ASSOCIATE, CASE, DO, FORALL, IF, SELECT TYPE, WHERE construct or an action statement.

executable statement

An instruction to perform or control one or more computational actions.

explicit coshape coarray

Explicit coshape coarray declares corank and cobounds. The entity is a nonallocatable coarray.

explicit initialization

Explicit initialization may be specified for objects of intrinsic or derived type in type declaration statements or DATA statements. An object of a derived type that specifies default initialization may not appear in a DATA statement.

explicit interface

For a procedure referenced in a scoping unit, the property of being an internal procedure, a module procedure, an intrinsic procedure, an external procedure that has an interface block, a recursive procedure reference in its own scoping unit, or a dummy procedure that has an interface block.

explicit-shape array

A named array that is declared with explicit bounds.

expression

A sequence of operands, operators, and parentheses. It may be a variable, a constant, a function reference, or may represent a computation.

extended type

An extensible type that is an extension of a different type. This type is declared having the EXTENDS attribute.

extensible type

A type that uses the EXTENDS attribute to derive a new type. It does not have the BIND attribute and is not sequence type.

extension type

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

extent

The size of one dimension of an array.

external association

Characteristic determining that a C language entity is global for a program.

external file

A sequence of records that exists in a medium external to the program.

external procedure

A procedure that is defined by an external subprogram or by a means other than Fortran.

external subprogram

A subprogram that is not in a main program, module, or another subprogram.

external unit

A mechanism that is used to refer to an external file. It is identified by a nonnegative INTEGER.

file

An internal file or an external file.

file storage unit

The storage unit of an unformatted file or a stream file.

final subroutine

The subroutine called automatically during finalization.

finalization

Refers to the calling of the user-defined final subroutine immediately before an object is destroyed.

function

A procedure that is invoked in an expression and components a value which is then used in evaluating the expression.

function result

The data object that returns the value of a function.

function subprogram

A sequence of statements beginning with a FUNCTION statement that is not in an interface block and ending with the corresponding END statement.

generic identifier

A lexical token that appears in an INTERFACE statement and is associated with all the procedures in the interface block, or a lexical token written in a GENERIC statement and related to all its specific type bound procedures.

generic interface

An interface specified by a generic procedure binding or a generic interface block.

generic interface block

An interface block that has a generic specification.

global entity

An entity identified by a lexical token whose scope is a program. It may be a program unit, a common block, or an external procedure.

host

A main program or subprogram that contains an internal subprogram is called the host of the internal subprogram. A module that contains a module subprogram is called the host of the module subprogram.

host association

The process by which an internal subprogram, module subprogram, or derived type definition accesses entities of its host, and the process by which an IMPORT statement in an interface body accesses named entities of its host.

host scoping unit

A scoping unit that immediately surrounds another scoping unit.

image index

An image index that identifies an image is an integer value in the range one to the number of images.

image selector

An image selector determines the image index for a coindexed object.

implied shape array

An implied shape array is an array named constant that takes its shape from the initialization expression in its declaration.

implicit interface

A procedure referenced in a scoping unit other than its own is said to have an implicit interface if the procedure is an external procedure that does not have an interface block, a dummy procedure that does not have an interface block, or a statement function.

inherit

Inheriting and acquiring from a parent. Type parameters, components, and procedure bindings that are automatically acquired from a parent type without being explicitly declared in an extended type are said to be inherited.

inheritance association

The relationship between an inherited component and the parent component in an extended type.

initialization expression

An expression satisfying rules that ensure that its value does not vary during program execution.

inquiry function

An intrinsic function or a function defined in an intrinsic module. The result is determined on the basis of the properties, rather than the values, of one or more arguments.

instance of a subprogram

The copy of a subprogram that is created when a procedure defined by the subprogram is invoked.

intent

An attribute of a dummy argument that is neither a procedure nor a pointer, which indicates whether it is used to transfer data into the procedure, out of the procedure, or both.

interface block

A sequence of statements from an INTERFACE statement to the corresponding END INTERFACE statement.

interface body

A sequence of statements in an interface block from a FUNCTION or SUBROUTINE statement to the corresponding END statement.

interface of a procedure

See procedure interface.

internal file

A CHARACTER variable that is used to transfer and convert data from internal storage to internal storage.

internal procedure

A procedure that is defined by an internal subprogram.

internal subprogram

A subprogram in a main program or another subprogram.

interoperable

The property guaranteeing that a C language entity that is equivalent to the Fortran language entity can be defined.

intrinsic

An adjective applied to data types, operations, assignment statements, procedures and modules that are defined in the standard and may be used in any scoping unit without further definition or specification.

invoke

To call a subroutine by a CALL statement or by a defined assignment statement.

To call a function by a reference to it by name or operator during the evaluation of an expression.

To call a final subroutine by means of finalization.

keyword

A word forming part of a statement syntax, or a name used to identify an item in a list.

kind type parameter

A parameter setting the value of the identification number of a kind prepared in one intrinsic type, or a derived type parameter declared to have the KIND attribute.

length of a character string

The number of characters in the character string.

lexical token

A sequence of one or more characters with a specified interpretation.

line

A sequence of 0 to 255 characters, which may contain Fortran statements, a comment, or an INCLUDE line.

literal constant

A constant without a name.

local entity

An entity identified by a lexical token whose scope is a scoping unit.

local variable

A local variable for a specific scope. A variable that is not captured by use association or host association, is not a dummy argument, and is not a variable in a common block.

main program

A program unit that is not a module, external subprogram, or block data program unit. Can be programmed by a means other than Fortran.

module

A program unit that contains or accesses definitions to be accessed by other program units.

module procedure

A procedure that is defined by a module subprogram.

module subprogram

A subprogram that is in a module but is not an internal subprogram.

name

A lexical token consisting of a letter or a '$' followed by up to 239 alphanumeric characters (letters, digits, and underscores) and '$'.

named

Having a name. That is, in a phrase such as "named variable," the word "named" signifies that the variable name is not qualified by a subscript list, substring specification, and so on. For example, if X is an array variable, the reference "X" is a named variable while the reference "X(1)" is a subobject designator.

named constant

A constant that has a name.

NaN

The value of not-a-number in IEEE arithmetic. It is generated by undefined values and invalid operations.

nonexecutable statement

A statement used to configure the program environment in which computational actions take place.

numeric storage unit

The unit of storage for holding a scalar that is not a pointer and is of type default REAL, default INTEGER, or default LOGICAL.

numeric type

INTEGER, REAL or COMPLEX type.

object

Data object.

obsolescent feature

A feature that is considered to have been redundant but that is still in frequent use.

operand

An expression that precedes or succeeds an operator.

operation

A computation involving one or two operands.

operator

A lexical token that specifies an operation.

override

When a procedure is bound to an extended type, that is given priority over the procedure that would be inherited from the parent type.

parent component

Corresponds to the inherited part of an extended type, and is the entity component of that type.

parent type

Extensible type of the derivation source of an extended type.

passed-object dummy argument

A dummy argument of a type bound procedure or of a procedure pointer component. It is associated with an object used to call that procedure.

pointer

A variable that has the POINTER attribute. A pointer shall not be referenced or defined unless it is pointer associated with a target. If it is an array, it does not have a shape unless it is pointer associated, although it does have a rank.

pointer assignment

The pointer association of a pointer with a target by the execution of a pointer assignment statement or the execution of an assignment statement for a data object of derived type having the pointer as a subobject.

pointer assignment statement

A statement of the form "*pointer-object* => *target*".

pointer associated

The relationship between a pointer and a target following a pointer assignment or a valid execution of an ALLOCATE statement.

pointer association

The process by which a pointer becomes pointer associated with a target.

polymorphic

The ability to have a different type during program execution. Objects declared with the keyword CLASS specified are polymorphic.

preconnected

A property describing a unit that is connected to an external file at the beginning of execution of a program. Such a unit may be specified in input/output statements without an OPEN statement being executed for that unit.

present

A dummy argument is present in an instance of a subprogram if it is associated with an actual argument and the actual argument is a dummy argument that is present in the invoking subprogram or is not a dummy argument of the invoking subprogram.

procedure

A computation that may be invoked during program execution. It may be a function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than one procedure if it contains ENTRY statements.

procedure designator

Designator of procedure.

procedure interface

An interface comprising procedure characteristics, procedure names, dummy argument names, and generic identifiers.

processor

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

processor dependent

The term for functionality that is not completely prescribed by standards. The processor determines the means and meaning of this type of functionality.

program

A set of program units that includes exactly on main program.

program unit

The fundamental component of a program. Sequences of statements, comments, and INCLUDE lines. It may be a main program, a module, an external subprogram, or a block data program unit.

pure procedure

A pure intrinsic procedure, a procedure defined by a pure subprogram, or a statement function that references only pure functions.

rank

The number of dimensions of an array. Zero for a scalar.

record

A sequence of values or characters that is treated as a whole within a file.

reference

The appearance of a data object name or subobject designator in a context requiring the value at that point during execution, the appearance of a procedure name, its operator symbol, or a defined assignment statement in a context requiring execution of the procedure at that point, or the appearance of a module in a USE statement.

result variable

The variable that returns the value of a function.

restricted expression

An expression that limits the use of specifications such as length type parameters and array shapes.

rounding mode

The method of selecting an operation result that cannot be represented exactly. IEEE arithmetic has four modes: rounding to zero, truncation, rounding up (to ∞), and rounding down (to $-\infty$). Two modes, COMPATIBLE and PROCESSOR_DEFINED are also available for input/output.

scalar

A single datum that is not an array.

Not having the property of being an array.

scope

That part of a program within which a lexical token has a single interpretation. It may be a program, a scoping unit, a construct, a single statement, or a part of a statement.

scoping unit

One of the following:

A derived-type definition,

An interface body, excluding any derived-type definitions and interface bodies in it, or

A program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms in it.

section subscript

A subscript, vector subscript, or subscript triplet in an array section selector.

selector

A syntactic mechanism for designating:

Part of a data object. It may designate a substring, an array element, an array section, or a structure component.

The set of values for which a CASE block is executed.

An object that uses type to determine which branch of a SELECT TYPE construct is executed.

An object associated with an associate name in an ASSOCIATE construct.

shape

For an array, the rank and extents. The shape may be represented by the rank-one array whose elements are the extents in each dimension.

size

For an array, the total number of elements.

statement

A sequence of lexical tokens. It usually consists of a single line, but a statement may be continued from one line to another and the semicolon symbol may be used to separate statements within a line.

statement entity

An entity identified by a lexical token whose scope is a single statement or part of a statement.

statement function

A procedure specified by a single statement that is similar in form to an assignment statement.

statement label

A lexical token consisting of up to five digits that precedes a statement and may be used to refer to the statement.

storage association

The relationship between two storage sequences if a storage unit of one is the same as a storage unit of the other.

storage sequence

A sequence of contiguous storage units.

storage unit

A character storage unit, a numeric storage unit, or an unspecified storage unit.

stride

The increment specified in a subscript triplet.

structure

A scalar data object of derived type.

structure component

Part of a derived type object.

structure constructor

A syntax mechanism used to construct a derived type value.

subobject

A portion of a named data object that may be referenced or defined independently of other portions. It may be an array element, an array section, a structure component, substrings, the real part of COMPLEX type objects, or the imaginary part of COMPLEX type objects.

subprogram

A function subprogram or a subroutine subprogram.

subroutine

A procedure that is invoked by a CALL statement or by a defined assignment statement.

subroutine subprogram

A sequence of statements beginning with a SUBROUTINE statement that is not in an interface block and ending with the corresponding END statement.

subscript

One of the lists of scalar INTEGER expressions in an array element selector.

subscript triplet

An item in the list of an array section selector that contains a colon and specifies a regular sequence of INTEGER values.

substring

A contiguous portion of a scalar character string. Note that an array section can include a substring selector; the result is called an array section and not a substring.

target

A named data object specified in a TARGET statement or in a type declaration statement containing the TARGET attribute, a data object created by an ALLOCATE statement for a pointer, or a subobject of such an object.

TKR compatible

Type compatible, same values as kind type parameters and same rank.

transformational function

An intrinsic function that is neither an elemental function nor an inquiry function. It usually has array arguments and an array result whose elements have values that depend on the values of many of the elements of the arguments.

type

Data type.

type bound procedure

A procedure binding within a type definition. The procedure can reference any object with that dynamic type using the binding name, as a user-defined operation, via a user-defined assignment, or as part of a finalization process.

type compatible

Refers to all data entities of a type compatible with other data entities of the same type. Unlimited polymorphic data entities are type compatible with all data entities. Polymorphic data entities that are not unlimited polymorphic are type compatible with data entities for which the dynamic type is an extended type of the declared type of that polymorphic data entity.

type declaration statement

An INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, TYPE(type-name), or BYTE statement.

type parameter

A parameter of an intrinsic data type. KIND and LEN are the type parameters. The type parameters of a derived type are defined within the derived type definition.

ultimate component

For a derived type or a structure, a component that is of intrinsic type, has the ALLOCATABLE attribute, or has the POINTER attribute, or an ultimate component of a component that is a derived type and does not have the POINTER attribute or the ALLOCATABLE attribute.

undefined

For a data object, the property of not having a determinate value.

unsigned

The modifier that prefixes a numeric data types in a C program. This indicates that the type is comprised of only nonnegative values. Fortran does not have anything similar.

use association

The association of names in different scoping units specified by a USE statement.

variable

A data object whose value can be defined and redefined during the execution of a program. It may be a named data object, an array element, an array section, a structure component, or a substring.

vector subscript

A section subscript that is an INTEGER expression of rank one.

whole array

A named array.

Appendix E ASCII Character Set

Fortran programs may use the full ASCII Character Set as listed below. The characters are listed in collating sequence from first to last. Characters preceded by up arrows (^) are ASCII Control Characters.

Table E.1 ASCII Chart

Character	HEX value	Decimal value	ASCII Abbr.	Description
^@	00	0	NUL	null<R>
^A	01	1	SOH	start of heading
^B	02	2	STX	start of text
^C	03	3	ETX	break, end of text
^D	04	4	EOT	end of transmission
^E	05	5	ENQ	enquiry
^F	06	6	ACK	acknowledge
^G	07	7	BEL	bell
^H	08	8	BS	backspace
^I	09	9	HT	horizontal tab
^J	0A	10	LF	line feed
^K	0B	11	VT	vertical tab
^L	0C	12	FF	form feed
^M	0D	13	CR	carriage return
^N	0E	14	SO	shift out
^O	0F	15	SI	shift in
^P	10	16	DLE	data link escape
^Q	11	17	DC1	device control 1
^R	12	18	DC2	device control 2
^S	13	19	DC3	device control 3
^T	14	20	DC4	device control 4
^U	15	21	NAK	negative acknowledge
^V	16	22	SYN	synchronous idle
^W	17	23	ETB	end of transmission block
^X	18	24	CAN	cancel
^Y	19	25	EM	end of medium
^Z	1A	26	SUB	end-of-file
^[	1B	27	ESC	escape
^\	1C	28	FS	file separator
^]	1D	29	GS	group separator
^^	1E	30	RS	record separator
^	1F	31	US	unit separator
	20	32	SP	space, blank
!	21	33	!	exclamation point
"	22	34	"	quotation mark

Character	HEX value	Decimal value	ASCII Abbr.	Description
#	23	35	#	number sign
$	24	36	$	dollar sign
%	25	37	%	percent sign
&	26	38	&	ampersand
'	27	39	'	apostrophe
(	28	40	(	left parenthesis
)	29	41	)	right parenthesis
*	2A	42	*	asterisk
+	2B	43	+	plus
,	2C	44	,	comma
-	2D	45	-	hyphen, minus
.	2E	46	.	period, decimal point
/	2F	47	/	slash, slant
0	30	48	0	zero
1	31	49	1	one
2	32	50	2	two
3	33	51	3	three
4	34	52	4	four
5	35	53	5	five
6	36	54	6	six
7	37	55	7	seven
8	38	56	8	eight
9	39	57	9	nine
:	3A	58	:	colon
;	3B	59	;	semicolon
<	3C	60	<	less than
=	3D	61	=	equal
>	3E	62	>	greater than
?	3F	63	?	question mark
@	40	64	@	commercial at sign
A	41	65	A	uppercase A
B	42	66	B	uppercase B
C	43	67	C	uppercase C
D	44	68	D	uppercase D
E	45	69	E	uppercase E
F	46	70	F	uppercase F
G	47	71	G	uppercase G
H	48	72	H	uppercase H
I	49	73	I	uppercase I

Character	HEX value	Decimal value	ASCII Abbr.	Description
J	4A	74	J	uppercase J
K	4B	75	K	uppercase K
L	4C	76	L	uppercase L
M	4D	77	M	uppercase M
N	4E	78	N	uppercase N
O	4F	79	O	uppercase O
P	50	80	P	uppercase P
Q	51	81	Q	uppercase Q
R	52	82	R	uppercase R
S	53	83	S	uppercase S
T	54	84	T	uppercase T
U	55	85	U	uppercase U
V	56	86	V	uppercase V
W	57	87	W	uppercase W
X	58	88	X	uppercase X
Y	59	89	Y	uppercase Y
Z	5A	90	Z	uppercase Z
[	5B	91	[	left bracket
\	5C	92	\	backslash
]	5D	93	]	right bracket
^	5E	94	^	up-arrow, circumflex, caret
_	5F	95	UND	back-arrow, underscore
`	60	96	GRA	grave accent
a	61	97	LCA	lowercase a
b	62	98	LCB	lowercase b
c	63	99	LCC	lowercase c
d	64	100	LCD	lowercase d
e	65	101	LCE	lowercase e
f	66	102	LCF	lowercase f
g	67	103	LCG	lowercase g
h	68	104	LCH	lowercase h
i	69	105	LCI	lowercase i
j	6A	106	LCJ	lowercase j
k	6B	107	LCK	lowercase k
l	6C	108	LCL	lowercase l
m	6D	109	LCM	lowercase m
n	6E	110	LCN	lowercase n
o	6F	111	LCO	lowercase o
p	70	112	LCP	lowercase p

Character	HEX value	Decimal value	ASCII Abbr.	Description
q	71	113	LCQ	lowercase q
r	72	114	LCR	lowercase r
s	73	115	LCS	lowercase s
t	74	116	LCT	lowercase t
u	75	117	LCU	lowercase u
v	76	118	LCV	lowercase v
w	77	119	LCW	lowercase w
x	78	120	LCX	lowercase x
y	79	121	LCY	lowercase y
z	7A	122	LCZ	lowercase z
{	7B	123	LBR	left brace
|	7C	124	VLN	vertical line
}	7D	125	RBR	right brace
~	7E	126	TIL	tilde
	7F	127	DEL,RO	delete, rubout

Appendix F Fortran 2003 Additional Specifications

The additional specifications are the following 70 items in the Fortran 2003 standard. The table below shows each support situation of Fujitsu Fortran System and the description in this manual.

Table F.1 Support situation of Fortran 2003 and the description in this manual.

	Specifications	Support situation	Description in this manual
1	Type extension	o	o
2	Enumerations	o	o
3	Structure constructor extension	o	o
4	Array constructor extension	o	o
5	Component, dummy argument, function result of allocatable attribute	o	o
6	Allocatable scalar	o	o
7	Deferred length type parameter of character type	o	o
8	Renaming operators on the USE statement	o	o
9	POINTER attribute and INTENT attribute	o	o
10	ASYNCHRONOUS attribute/statement	o	o
11	Remove of restriction for module procedure of PRIVATE type	o	o
12	PROTECTED attribute/statement	o	o
13	PUBLIC/PRIVATE attribute of derived type component	o	o
14	VOLATILE attribute/statement	o	o
15	VALUE attribute/statement	o	o
16	IMPORT statement	o	o
17	PROCEDURE statement in interface	o	o
18	Abstract interface	o	o
19	Procedure declaration statement(PROCEDURE statement)	o	o
20	Declaration and specification of procedure pointer	o	o
21	Assignment and reference of procedure pointer	o	o
22	ALLOCATE statement extension(ERRMSG=,SOURCE=)	o	o
23	Modify specification of assignment for allocatable array	o	o
24	Transferring an allocation (MOVE_ALLOC intrinsic subroutine)	o	o
25	ISO_FORTRAN_ENV intrinsic module	o	o
26	Intrinsic procedure get an environment/command line	o	o
27	Length extension of names and statements	o	o
28	Binary, octal and hex constants of intrinsic procedure argument	o	o
29	Remove some restrictions of specification and initialization expression	o	o
30	Complex constants extension	o	o
31	Intrinsic functions argument extension	o	o
32	Lower specification and remapping of pointer assignment	o	o
33	Asynchronous input/output statement	o	o
34	FLUSH statement	o	o

	Specifications	Support situation	Description in this manual
35	IOMSG= specifier	o	o
36	Stream access input/output	o	o
37	ROUND= specifier	o	o
38	DECIMAL= specifier	o	o
39	SIGN= specifier	o	o
40	Kind type parameter extension of integer specifier	o	o
41	Intrinsic procedure for input/output (IS_IOSTAT_END, etc)	o	o
42	Derived type input/output	x	o
43	Input/output of IEEE exception value	o	o
44	Comma after a P edit descriptor	o	o
45	Pointer or allocatable of namelist group object	o	o
46	IEEE intrinsic module (IEEE_EXCEPTIONS,IEEE_ARITHMETIC,IEEE_FEATURES)	o	o
47	IEEE exceptions	o	o
48	IEEE rounding modes	o	o
49	IEEE halting	o	o
50	IEEE floating point status	o	o
51	IEEE exceptional values	o	o
52	IEEE arithmetic	o	o
53	IEEE module procedure	o	o
54	Intrinsic module ISO_C_BINDING	o	(*)
55	Interoperability of intrinsic types	o	(*)
56	Interoperability with C pointers	o	(*)
57	Interoperability of derived types	o	(*)
58	Interoperability of variables	o	(*)
59	Interoperability of procedures	o	(*)
60	Interoperability of global data	o	(*)
61	Structure constructors and generic names	o	o
62	Parameterized derived types	o	o
63	Final subroutine	o	o
64	Type-bound procedures	o	o
65	PASS attribute	o	o
66	ASSOCIATE construct	o	o
67	Polymorphic (CLASS)	o	o
68	SELECT TYPE construct	o	o
69	Deferred bindings and abstract types	o	o
70	International character sets	x	o

o: Support or description

x: No support

*) See "Chapter 11 Mixed Language Programming in Fortran User's Guide".

Appendix G Limitations

The table below shows each limitation for the Fortran 2003 standard.

Table G.1 Fortran 2003 Standard Limitations.

	Limitation specifications	Detail limitations
1	Derived-type input / output	DT edit descriptor and user-defined derived-type input/output cannot specify.
2	Universal character type	ISO_10646 character(Universal character) type cannot specify.

Appendix H Support Fortran 2008 Specifications

Fujitsu Fortran System supports a part of specifications in Fortran 2008.
The table below shows supported Fortran 2008 specifications in Fujitsu Fortran System.

Table H.1 Supported Fortran 2008 specifications

[COARRAY specifications] - COARRAY Explicit coshape coarray Allocatable coarray - CODIMENSION attribute, CODIMENSION statement - Coarray specifier in ALLOCATABLE statement, TARGET statement, type declaration statement - Image selector - Execution statements Specify coarray in ALLOCATE or DEALLOCATE statement CRITICAL construct, END CRITICAL statement, LOCK statement, SYNC ALL statement, SYNC IMAGE statement, SYNC MEMORY statement, UNLOCK statement - Intrinsic procedures ATOMIC_DEFINE, ATOMIC_REF, CO_MAX, CO_MIN, CO_SUM, IMAGE_INDEX, LCOBOUND, NUM_IMAGES, THIS_IMAGE, UCOBOUND Specify coarray in MOVE_ALLOC - ISO_FORTRAN_ENV intrinsic module LOCK_TYPE type, ATOMIC_INT_KIND, ATOMIC_LOGICAL_KIND, STAT_LOCKED, STAT_LOCKED_OTHER_IMAGE, STAT_STOPPED_IMAGE, STAT_UNLOCKED
[Executable Statements] - DO CONCURRENT construct - ERROR STOP statement
[Nonexecutable Statements] - CONTIGUOUS statement
[Data declaration or specification] - Implied shape array - Double colon after PROCEDURE in interface block
[Attribute] - CONTIGUOUS attribute - Implicit SAVE attribute declaration part of module
[Intrinsic procedures] - Generic intrinsic function ACOSH, ASINH, ATANH, BGE, BGT, BLE, BLT, DSHIFTL, DSHIFTR, HYPOT,LEADZ, MASKL, MASKR, MERGE_BITS, POPCNT, POPPAR, SHIFTA, SHIFTL,SHIFTR, STORAGE_SIZE, TRAILZ, IS_CONTIGUOUS - Intrinsic function that has complex type argument ACOS, ASIN, ATAN, COSH, SINH, TAN, TANH

- Intrinsic function ATAN that has two arguments.

[Intrinsic module function]

- C_SIZEOF
- COMPILER_OPTIONS
- COMPILER_VERSION

[Construct]

- BLOCK construct, END BLOCK statement

[Specific binding]

- Multiple procedures can be specified in specific type binding declarations.

[ISO_FORTRAN_ENV Intrinsic Module]

- INT8
- INT16
- INT32
- INT64
- REAL32
- REAL64
- REAL128
- CHARACTER_KINDS
- INTEGER_KINDS
- LOGICAL_KINDS
- REAL_KINDS

[ALLOCATE Statement]

- The allocate shape specifications list can be omitted if a SOURCE= specifier or MOLD= specifier is specified.
- MOLD= specifier

[Pointer assignment]

- The pointer target is simply contiguous without of rank one when a bound remapping list is specified.

[Optional dummy argument]

- The optional dummy argument is nonpointer, nonallocatable, and the corresponding actual argument is an unallocated or disassociated status.

[Stop Statement]

- The scalar default CHARACTER initialization expression, scalar INTEGER initialization expression.

Index

[C]

[D]

[E]

[F]

[G]

[H]

[I]

[R]

[S]

[T]

[U]

[V]

[W]

[X]