LF Fortran 95 Language Reference

Revision G.02

Lahey Computer Systems, Inc.
865 Tahoe Boulevard
P.O. Box 6091
Incline Village, NV 89450-6091
(775) 831-2500
Fax: (775) 831-8123

http://www.lahey.com

Technical Support
support2@lahey.com (all versions)

Table of Contents

Introduction

Lahey/Fujitsu Fortran 95 (LF95) is a complete implementation of the Fortran 95 standard. Numerous popular extensions are supported.

This manual is intended as a reference to the Fortran 95 language for programmers with experience in Fortran. For information on creating programs using the LF95 Language System, see the Lahey/Fujitsu Fortran 95 User's Guide.

Manual Organization

The manual is organized in eight parts:

- Chapter 1, *Elements of Fortran*, 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*, gives detailed syntax and constraints for Fortran statements, constructs, and intrinsic procedures.
- Appendix A, *Fortran 77 Compatibility*, discusses issues of concern to programmers who are compiling their Fortran 77 code with LF95.
- Appendix B, *New in Fortran 95*, lists Fortran 95 features that were not part of standard Fortran 77.
- Appendix C, *Intrinsic Procedures*, is a table containing brief descriptions and specific names of procedures included with LF95.
- Appendix D, *Porting Extensions*, lists the various non-standard features provided to facilitate porting from other systems.
- Appendix E, *Glossary*, defines various technical terms used in this manual.
- Appendix F, *ASCII Chart*, details the 128 characters of the ASCII set.

Notational Conventions

The following conventions are used throughout the manual:

blue text	indicates an extension to the Fortran 95 standard.
`code`	is indicated by courier font.

In syntax descriptions, *[brackets]* enclose optional items. An ellipsis, “...”, following an item indicates that more items of the same form may appear. *Italics* indicate text to be replaced by you. Non-italic letters in syntax descriptions are to be entered exactly as they appear.

1 Elements of Fortran

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 ‘`_`’.

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

‘`?`’ and ‘`$`’ have no special meaning.

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.

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 ‘`$`’ and consists entirely of letters, digits, underscores, and the ‘`$`’ character. A standard conforming name must be 31 characters or less in length, but LF95 accepts names of up to 240 characters in length.

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
name_toooooooooooooooooooooooooooooooooooooooooooooooooooo&
&ooooooooooooooooooooooooooooooooooooooooooooooooooooooooo&
&ooooooooooooooooooooooooooooooooooooooooooooooooooooooooo&
&ooooooooooooooooooooooooooooooooooooooooooooooooooooooooo&
&ooooooooo_long
illegal_@_character
```

Statement Labels

Fortran statements can have labels consisting of one to five digits, at least one of which is non-zero. Leading zeros are not significant in distinguishing statement labels. The following labels are valid:

```
123
5000
10000
1
0001
```

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

Source Form

Fortran offers two source forms: fixed and free.

Fixed Source Form

Fixed source form is the traditional Fortran source form and is based on the columns of a punched card. There are restrictions on where statements and labels can appear on a line. Except in CHARACTER literals, blanks are ignored.

Except within a comment:

- Columns 1 through 5 are reserved for statement labels. Labels can contain blanks.
- 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 up to 19 continuation lines. Continuation lines must not be labeled.
- Columns 7 through 72 are used for Fortran statements.
- Columns after 72 are ignored.

Fixed source form comments are formed by beginning a line with a '`C`' or a '`*`' in column 1. Additionally, trailing comments can be formed by placing a '`!`' in any column except column 6. A '`!`' in a CHARACTER literal does not indicate a trailing comment. Comment lines must not be continued, but a continuation line can contain a trailing comment. An END statement must not be continued.

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

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 132 characters long. Blanks are used to separate names, constants, or labels from adjacent names, constants, or labels. Blanks are also used to separate Fortran keywords, with the following exceptions, for which the blank separator is optional:

- BLOCK DATA
- DOUBLE PRECISION
- ELSE IF
- END BLOCK DATA
- END DO
- END FILE
- END FUNCTION
- END IF
- END INTERFACE
- END MODULE
- END PROGRAM
- END SELECT
- END SUBROUTINE
- END TYPE
- END WHERE
- GO TO
- IN OUT
- SELECT CASE

The '`!`' character begins a comment except when it appears in a CHARACTER literal. 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 literal 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 must be the '`&`' character followed by successive characters of the name, constant, keyword, or label. If a CHARACTER literal is to be continued, the '`&`' character ending the line cannot be followed by a trailing comment. A free source form statement can have up to 39 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.

Data

Fortran offers the programmer a variety of ways to store and refer to data. You can refer to data literally, as in the real numbers `4.73` and `6.23E5`, the integers `-3000` and `65536`, or the CHARACTER literal `"Continue (y/n)?"`. Or, you can store and reference variable data, using names such as `x` or `y`, `DISTANCE_FROM_ORIGIN` or `USER_NAME`. Constants such as pi or the speed of light can be given names and constant values. You can store data in a fixed-size area in memory, or allocate memory as the program needs it. Finally, Fortran offers various means of creating, storing, and referring to structured data, through use of arrays, pointers, and derived types.

Intrinsic Data Types

The five intrinsic data types are INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER. The DOUBLE PRECISION data type available in Fortran 77 is still supported, but is considered a subset, or kind, of the REAL data type.

Kind

In Fortran, an intrinsic data type has one or more *kinds*. In LF95 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. Two intrinsic inquiry functions, SELECTED_INT_KIND and SELECTED_REAL_KIND, 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 90 or 95 systems. The kinds available in LF95 are summarized in the following table:

Table 1: Intrinsic Data Types

Type	Kind Type Parameter	Notes
INTEGER	1	Range: -128 to 127
INTEGER	2	Range: -32,768 to 32,767
INTEGER	4*	Range: -2,147,483,648 to 2,147,483,647
INTEGER	8	Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
REAL	4*	Range: $1.18 * 10^{-38}$ to $3.40 * 10^{38}$ Precision: 6-7 decimal digits
REAL	8	Range: $2.23 * 10^{-308}$ to $1.79 * 10^{308}$ Precision: 15-16 decimal digits
REAL	16	Range: 10^{-4931} to 10^{4932} Precision: approximately 33 decimal digits
COMPLEX	4*	Range: $1.18 * 10^{-38}$ to $3.40 * 10^{38}$ Precision: 7-8 decimal digits
COMPLEX	8	Range: $2.23 * 10^{-308}$ to $1.79 * 10^{308}$ Precision: 15-16 decimal digits
COMPLEX	16	Range: 10^{-4931} to 10^{4932} Precision: approximately 33 decimal digits
LOGICAL	1	Values: .TRUE. and .FALSE.
LOGICAL	4*	Values: .TRUE. and .FALSE.
CHARACTER	1*	ASCII character set

* default kinds

Length

The number of characters in a CHARACTER data object is indicated by its *length type parameter*. For example, the CHARACTER literal `"Half Marathon"` has a length of thirteen.

Literal Data

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.

INTEGER literals

An INTEGER literal 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 default kind. Examples of valid INTEGER literals are

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

`34` and `-256` are of type default INTEGER. `345_4` is an INTEGER of kind `4` (default INTEGER in LF95). In the last example, `mykind` must have been previously declared as a scalar INTEGER named constant with the value of an INTEGER kind type parameter (`1`, `2`, or `4` in LF95).

A binary, octal, or hexadecimal constant can appear in a DATA statement. Such constants are formed by enclosing a series of binary, octal, or hexadecimal digits in apostrophes or quotation marks, and preceding the opening apostrophe or quotation mark with a `B`, `O`, or `Z` for binary, octal, and hexadecimal representations, respectively. Two valid examples are

```
B'10101'    Z"1AC3"
```

REAL literals

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, or `Q` for quad precision. If the optional underscore and kind type parameter are not present, the REAL literal is of default kind. Examples of valid REAL literals are

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

The first three examples are of type default REAL. The last example is a REAL of kind `8`.

COMPLEX literals

A COMPLEX literal is formed by enclosing in parentheses a comma-separated pair of REAL or INTEGER literals. The first of the REAL or INTEGER literals represents the real part of the complex number; the second represents the imaginary part. The kind type parameter of a COMPLEX 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 literals are

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

The first three examples are of default kind, where four bytes are used to represent each part, real or imaginary, of the complex number. The fourth example uses eight bytes for each part.

LOGICAL literals

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 default kind. Examples of valid LOGICAL literals are:

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

In the last example, `mykind` must have been previously declared as a scalar INTEGER named constant with the value of a LOGICAL kind type parameter (`1` or `4` in LF95). The first two examples are of type default LOGICAL.

CHARACTER literals

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 the ship!'
ASCII_'foobeedoodah'
""
''
```

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

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.

Once a data object has a name, it can be accessed in its entirety by referring to that name. For some data objects, such as character strings, arrays, and derived types, portions of the data object can also be accessed directly. In addition, aliases for a data object or a portion of a data object, known as pointers, can be established and referred to.

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 INTEGER data objects and the other letters begin REAL data objects. 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.

Type Declaration Statements

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 (and DOUBLE PRECISION), COMPLEX, LOGICAL, or CHARACTER, as well as for derived types (see *"Derived Types"* on page 16).

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:

- DIMENSION — the data object is an array (see *"DIMENSION Statement"* on page 114).
- PARAMETER — the data object is a named constant (see *"PARAMETER Statement"* on page 214).
- POINTER — the data object is to be used as an alias for another data object of the same type, kind, and rank (see *"POINTER Statement"* on page 217).
- TARGET — the data object that is to be aliased by a POINTER data object (see *"TARGET Statement"* on page 255).
- EXTERNAL — the name is that of an external procedure (see *"EXTERNAL Statement"* on page 135).

- ALLOCATABLE — the data object is an array that is not of fixed size, but is to have memory allocated for it as specified during execution of the program (see *"ALLOCATABLE Statement"* on page 66).

- INTENT(IN) — the dummy argument will not change in the subprogram
- INTENT(OUT) — the dummy argument is undefined on entry to the subprogram
- INTENT(IN OUT) — the dummy argument has an initial value on entry and may be redefined within the subprogram (see *"INTENT Statement"* on page 168).

- PUBLIC — the named data object or procedure in a MODULE program unit is accessible in a program unit that uses that module (see *"PUBLIC Statement"* on page 224).

- PRIVATE — the named data object or procedure in a MODULE program unit is accessible only in the current module (see *"PRIVATE Statement"* on page 221).

- INTRINSIC — the name is that of an intrinsic function (see *"INTRINSIC Statement"* on page 175).

- OPTIONAL — the dummy argument need not have a corresponding actual argument in a reference to the procedure in which the dummy argument appears (see *"OPTIONAL Statement"* on page 212).

- SAVE — the data object retains its value, association status, and allocation status after a RETURN or END statement (see *"SAVE Statement"* on page 237).

- SEQUENCE — the order of the component definitions in a derived-type definition is the storage sequence for objects of that type (see *"SEQUENCE Statement"* on page 242).

- VOLATILE — the data object may be referenced, become redefined or undefined by means not specified in the Fortran standard (see *"VOLATILE Statement"* on page 267).

- DLL_EXPORT (Windows only) — the name is an external procedure, or a common block name, that to be a DLL (see *"DLL_EXPORT Statement"* on page 115).

- DLL_IMPORT (Windows only) — the name is an external procedure, or a common block name, that uses a DLL (see *"DLL_IMPORT Statement"* on page 116).

- ML_EXTERNAL (Windows only) — the name is an external procedure, or a common block name, that is available for calling from a mixed language procedure (see *"ML_EXTERNAL Statement"* on page 199).

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 (*[lower-bound]* : *[upper-bound]*)

Where:

string is a string name or a CHARACTER literal.

lower-bound is the lower bound of a substring of *string*.

upper-bound is the upper bound of a substring of *string*.

If absent, *lower-bound* and *upper-bound* are given the values one and the length of the string, respectively. A substring has a length of zero if *lower-bound* is greater than *upper-bound*. *lower-bound* must not be less than one.

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

`abc_string(2:4)` is `“bcd”`

`abc_string(2:)` is `“bcdefg”`

`abc_string(:5)` is `“abcde”`

`abc_string(:)` is `“abcdefg”`

`abc_string(3:3)` is `“c”`

`“abcdef”(2:4)` is `“bcd”`

`“abcdef”(3:2)` is a zero-length string

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. A data object that is not an array is a *scalar*. Arrays can be specified by using the DIMENSION statement or by using the DIMENSION attribute in a type declaration statement. An array has a *rank* that is equal to the number of dimensions in the array; a scalar has rank zero. 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.

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:

array [(subscript-list)]

Where:

array is the name of the array.

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 *[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.

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, it is to an *array section* (see *"Array Sections"* on page 12).

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:

$$(1 + (s_1 - j_1)) + ((s_2 - j_2) \times d_1) + \ldots + ((s_n - j_n) \times d_{n-1} \times d_{n-2} \ldots \times d_1)$$

Where:

s_i is the subscript in the *i*th dimension.

j_i is the lower bound of the *i*th dimension.

d_i is the size of the *i*th dimension.

n is the rank of the array.

Another way of describing array element order is that the subscript of the leftmost dimension changes most rapidly as one goes from first element to last in array element order. 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.

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 (see the examples under *"Subscript Triplets"* and *"Vector Subscripts"* below). Note that an array section with only one element is not a scalar.

Subscript Triplets

The three components of a subscript triplet are the lower bound of the array section, the upper bound, and the stride (the increment between successive subscripts in the sequence), respectively. 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
```

Vector Subscripts

A vector (one-dimensional array) subscript can be used to refer to a section of a whole array. 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).

Arrays and Substrings

A CHARACTER array section or array element can have a substring specifier following the subscript list. If a whole array or an array section has a substring specifier, then the reference is an array section. For example,

```
character (len=10), dimension (10,10) :: my_string
my_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 `my_string`.

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. An *array pointer* assumes the shape of its target when used in a pointer assignment statement (see *"Allocatable Arrays"* on page 13 and *"Array Pointers"* on page 13). Allocatable arrays and array pointers together are known as *deferred-shape arrays*.
- A dummy array can assume a size and shape based on the size and shape of the corresponding actual argument (see *"Assumed-Shape Arrays"* on page 14).
- A dummy array can be of undeclared size (*"Assumed-Size Arrays"* on page 14).
- An array can have variable dimensions based on the values of dummy arguments (*"Adjustable and Automatic Arrays"* on page 15).

Allocatable Arrays

The ALLOCATABLE attribute can be given to an array in a type declaration statement or in an ALLOCATABLE statement. An allocatable array must be declared with the deferred-shape specifier, ':', for each dimension. For example,

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

declares two allocatable arrays, one of rank one and the other of 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 and size three and an array of rank three 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.

Array Pointers

The POINTER attribute can be given to an array in a type declaration statement or in a POINTER statement. An array pointer, like an allocatable array, is declared with the *deferred-shape specifier*, ':', for each dimension. 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, target, dimension(2,4) :: d
integer, pointer, dimension(:,:) :: c

c => d
```

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

Assumed-Shape Arrays

An *assumed-shape array* is a dummy array that assumes the shape of the corresponding actual argument. The lower bound of an assumed-shape array can be declared and can be different from that of the actual argument array. An assumed-shape specification is

[lower-bound] :

for each dimension of the assumed-shape array. For example

```
...
integer :: a(3,4)
...
call zee(a)
...

subroutine zee(x)
implicit none
integer, dimension(-1:,:) :: x
...
```

Here the dummy array `x` assumes the shape of the actual argument `a` with a new lower bound for dimension one.

The interface for an assumed-shape array must be explicit (see *"Explicit Interfaces"* on page 51).

Assumed-Size Arrays

An *assumed-size array* is a dummy array that's size is not known. All bounds except the upper bound of the last dimension are specified in the declaration of the dummy array. In the declaration, the upper bound of the last dimension is an asterisk. The two arrays have the same initial array element, and are storage associated.

You must not refer to an assumed-size array in a context where the shape of the array must 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)
...

subroutine zee(x)
integer, dimension(-1:*) :: x
...
```

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

Adjustable and Automatic Arrays

You can establish the shape of an array based on the values of dummy arguments. If such an array is a dummy array, it is called an *adjustable array*. If the array is not a dummy array it is called an *automatic array*. Consider the following example:

```
integer function bar(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`. `i` is an adjustable array and `j` is an automatic array.

Array Constructors

An array constructor is an unnamed array.

Syntax:

(/ *constructor-values* /)

Where:

constructor-values is a comma-separated list of
expr
or *ac-implied-do*

expr is an expression.

ac-implied-do is (*constructor-values, ac-implied-do-control*)

ac-implied-do-control is *do-variable* = *do-expr, do-expr [, do-expr]*

do-variable is a scalar INTEGER variable.

do-expr is a scalar INTEGER expression.

An array constructor is a rank-one array. If a constructor element is itself array-valued, the values of the elements, in array-element order, specify the corresponding sequence of elements of the array constructor. If a constructor value is an implied-do, it is expanded to form a sequence of values under the control of the *do-variable* as in the DO construct (see *"DO Construct"* on page 116).

```
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

```
real,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`.

Derived Types

Derived types are user-defined data types based on the intrinsic types, INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER. Where an array is a set of data all of the same type, a derived type can be composed of a combination of intrinsic types or other derived types. A data object of derived type is called a structure.

Derived-Type Definition

A derived type must be defined before objects of the derived type can be declared. A derived type definition specifies the name of the new derived type and the names and types of its components.

Syntax:

derived-type-statement
[private-sequence-statement]
type-definition-statement
[type-definition-statement]
...
END TYPE *[type-name]*

Where:

derived-type-statement is a derived type statement.

private-sequence-statement is a PRIVATE statement.
or a SEQUENCE statement.

type-definition-statement is an INTEGER, REAL, COMPLEX, DOUBLE PRECISION, LOGICAL, CHARACTER or TYPE statement.

A type definition statement in a derived type definition can have only the POINTER and DIMENSION attributes. It cannot be a function. It can be given a default initialization value, in which case the component acquires the SAVE attribute. A component array must be a deferred-shape array if the POINTER attribute is present, otherwise it must have an explicit shape.

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

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

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

In the example, type `coordinates` is a derived type with two REAL components: `latitude` and `longitude`. 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. In this way, linked lists, trees, and graphs can be formed.

There are two ways to use a derived type in more than one program unit. The preferred way is to define the derived type in a module (see *"Module Program Units"* on page 56) and use the module wherever the derived type is needed. Another method, avoiding modules, is to use a SEQUENCE statement in the derived type definition, and to define the derived type in exactly the same way in each program unit the type is used. This could be done using an include file. Components of a derived type can be made inaccessible to other program units by using a PRIVATE statement before any component definition statements.

Declaring Variables of Derived Type

Variables of derived type are declared with the TYPE statement. 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
```

Component References

Components of a structure are referenced using the percent sign '`%`' operator. For example, `latitude` in the structure `my_coordinates`, above, is referenced as `my_coordinates%latitude`. `latitude` in type `coordinates` in structure `my_town` is referenced as `my_town%coordinates%latitude`. If the variable is an array of structures, as in `cities`, above, array sections can be referenced, such as

```
cities(:,:)%name
```

which references the component `name` for all elements of `cities`, and

```
cities(1,1:2)%coordinates%latitude
```

which references element `latitude` of type `coordinates` for elements `(1,1)` and `(1,2)` only of `cities`. Note that in the first example, the syntax

```
cities%name
```

is equivalent and is an array section.

Structure Constructors

A structure constructor is an unnamed structure.

Syntax:

type-name (*expr-list*)

Where:

type-name is the name of the derived type.

expr-list is a list of expressions.

Each expression in *expr-list* must agree in number and order with the corresponding components of the derived type. Where necessary, intrinsic type conversions are performed. For non-pointer components, the shape of the expression must agree with that of the component.

```
type mytype            ! derived-type definition
        integer :: i,j
        character(len=40) :: string
end type mytype

type (mytype) :: a     ! derived-type declaration
a = mytype (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.

Pointers

In Fortran, a *pointer* is an alias. The variable it aliases is its *target*. Pointer variables must have the POINTER attribute; target variables must have either the TARGET attribute or the POINTER attribute.

Associating a Pointer with a Target

A pointer can only be associated with a variable that has the TARGET attribute or the POINTER attribute. Such an association can be made in one of two ways:

- explicitly with a pointer assignment statement.
- implicitly with an ALLOCATE statement.

Once an association between pointer and target has been made, any reference to the pointer applies to the target.

Declaring Pointers and Targets

A variable can be declared to have the POINTER or TARGET attribute in a type declaration statement or in a POINTER or TARGET statement. When declaring an array to be a pointer, you must declare the array with a deferred shape.

Example:

```
integer, pointer :: a, b(:,:)
integer, target :: c
a => c              ! pointer assignment statement
                    ! a is an alias for 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.

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
"to be" // ' or not to be' // text(1:23)
(-b + (b**2-4*a*c)**.5) / (2*a)
b%a - a(1:1000:10)
sin(a) .le.  .5
l .my_binary_operator.  r + .my_unary_operator.  m
```

The last example uses defined operations (see *“Defined Operations”* on page 53).

All array-valued operands in an expression must 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, described below. If there are multiple contiguous operations of the same precedence, subtraction and division are evaluated from left to right, exponentiation is evaluated from right to left, and other operations can be evaluated either way, depending on how the compiler optimizes the expression. Parentheses can be used to enforce a particular order of evaluation.

A *specification expression* is a scalar INTEGER expression that can be evaluated on entry to the program unit at the time of execution. An *initialization expression* is an expression that can be evaluated at compile time.

Intrinsic Operations

The intrinsic operators, in descending order of precedence are:

Table 2: Intrinsic Operators

Operator	Represents	Operands
**	exponentiation	two numeric
* 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 <= .GT. and > .GE. and >=	equal to not equal to less than less than or equal to greater than greater than or equal to	two numeric or two CHARACTER ———— two non-COMPLEX 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

Note: all operators within a given cell in the table are of equal precedence

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.

If an operation is performed on numeric or LOGICAL operands of the same type but different kind, the result has the kind of the operand offering the greater precision.

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

INTEGER Division

The result of a division operation between two INTEGER operands is the integer closest to the mathematical quotient and between zero and the mathematical quotient, inclusive. For example, `7/5` evaluates to `1` and `-7/5` evaluates to `-1`.

Input/Output

Fortran input and output are performed on logical *units*. A unit is

- a non-negative INTEGER associated with a physical device such as a disk file, the console, or a printer. The unit must be connected to a file or device in an OPEN statement, except in the case of pre-connected files.
- an asterisk, '*', indicating the standard input and standard output devices, usually the keyboard and monitor, that are preconnected.
- a CHARACTER variable corresponding to the name of an internal file.

Fortran statements are available to connect (OPEN) or disconnect (CLOSE) files and devices from input/output units; transfer data (PRINT, READ, WRITE); establish the position within a file (REWIND, BACKSPACE, ENDFILE); and inquire about a file or device or its connection (INQUIRE).

Pre-Connected Input/Output Units

Input/output units 5, 6 and * are automatically connected when used. Unit 5 is connected to the standard input device, usually the keyboard, and unit 6 is connected to the standard output device, usually the monitor. Unit * is always connected to the standard input and standard output devices.

Files

Fortran treats all physical devices, such as disk files, the console, printers, and internal files, as files. A file is a sequence of zero or more records. The data format (either formatted or unformatted), file access type (either direct or sequential) and record length determine the structure of the file.

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 must be repositioned with a REWIND or BACKSPACE statement before it is read from or written to again.

For non-advancing (partial record) input/output, if there is no error and no end-of-file condition, but an end-of-record condition occurs, the file is positioned after the record just read. If there is no end-of-record condition the file position is unchanged.

File Types

The type of file to be accessed is specified in the OPEN statement using the FORM= and ACCESS= specifiers (see *“OPEN Statement”* on page 209).

Formatted Sequential

- variable-length records terminated by end of line
- stored as CHARACTER data
- can be used with devices or disk files
- records must be processed in order
- files can be printed or displayed easily
- usually slowest

Formatted Direct

- fixed-length records - no header
- stored as CHARACTER data
- disk files only
- records can be accessed in any order
- not easily processed outside of LF95
- same speed as formatted sequential disk files

Unformatted Sequential

- variable length records separated by record marker
- stored as binary data
- disk files only
- records must be processed in order
- faster than formatted files
- not easily read outside of LF95

Unformatted Direct

- fixed-length records - no header
- stored as binary data
- disk files only
- records can be accessed in any order
- fastest
- not easily read outside of LF95

Binary (or Transparent)

- stored as binary data without record markers or header
- record length one byte but end-of-record restrictions do not apply
- records can be processed in any order
- can be used with disk files or other physical devices
- good for files that are accessed outside of LF95
- fast and compact

See *“File Formats”* in the User's Guide for more information.

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. This feature allows conversion from internal representation (binary, unformatted) to external representation (ASCII, formatted) without transferring data to an external device.

Carriage Control

The first character of a formatted record sent to a terminal device, such as the console or a printer, is used for carriage control and is not printed. The remaining characters are printed on one line beginning at the left margin. The carriage control character is interpreted as follows:

Table 3: Carriage Control

Character	Vertical Spacing Before Printing
0	Two Lines
1	To First Line of Next Page
+	None
Blank or Any Other Character	One Line

Input/Output Editing

Fortran provides extensive capabilities for formatting, or editing, of data. The editing can be explicit, using a *format specification*; or implicit, using list-directed input/output, in which data are edited using a predetermined format (see *"List-Directed Formatting"* on page 31). A *format specification* is a default CHARACTER expression and can appear

- directly as the FMT= specifier value.
- in a FORMAT statement whose label is the FMT= specifier value.
- in a FORMAT statement whose label was assigned to a scalar default INTEGER variable that appears as the FMT= specifier value.

The syntax for a format specification is

(*[format-items]*)

where *format-items* includes editing information in the form of *edit descriptors*. See *"FORMAT Statement"* on page 139 for detailed syntax.

Format Control

A correspondence is established between a format specification and items in a READ, WRITE or PRINT statement's input/output list in which the edit descriptors and input/output list are both interpreted from left to right. Each effective edit descriptor is applied to the corresponding data entity in the input/output list. Each instance of a repeated edit descriptor is an edit descriptor in effect. Three exceptions to this rule are

1. COMPLEX items in the input/output list require the interpretation of two F, E, EN, ES, D or G edit descriptors.
2. Control and character string edit descriptors do not correspond to items in the input/output list.
3. If the end of a complete format is encountered and there are remaining items in the input/output list, format control reverts to the beginning of the format item terminated by the last preceding right parenthesis, if it exists, and to the beginning of the format otherwise. If format control reverts to a parenthesis preceded by a repeat specification, the repeat specification is reused.

Data Edit Descriptors

Data edit descriptors control conversion of data to or from its internal representation.

Numeric Editing

The I, B, O, Z, Q, F, E, EN, ES, D, and G 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 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.

INTEGER Editing (I, B, O, and Z)

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 the manner of editing for INTEGER data. The *w* indicates the width of the field on input, including a sign (if present). The *m* indicates the minimum number of digits on output; *m* must not exceed *w* unless *w* is zero. The output width is padded with blanks if the number is smaller than the field, unless *w* is zero. If *w* is zero then a suitable width will be used to show all digits without any padding blanks. Note that an input width must always be specified.

REAL Editing (Q, F, D, and E)

The Q*w.d*, F*w.d*, E*w.d*, D*w.d*, E*w.d*E*e*, EN, and ES edit descriptors indicate the manner of editing of REAL and COMPLEX data.

Q, F, D, E, EN, and ES 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 field consists of an optional sign, followed by one or more digits that can contain a decimal point. If the decimal point is omitted, the rightmost *d* digits are interpreted as the fractional part. An exponent can be included in one of the following forms:

- An explicitly signed INTEGER constant.
- Q, E, or D 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 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 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 Q, E, and D editing, the output field consists of the following, in order:

1. zero or more blanks
2. a minus or an optional plus sign (see S, SP, and SS Editing)
3. a zero (depending on scale factor, see P Editing)
4. a decimal point
5. the *d* most significant digits, rounded
6. a Q, E, or a D
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.

For Q, E, and D editing, the scale factor *k* controls the position of the decimal point. If $-d < k \leq 0$, 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.

EN Editing

The EN edit descriptor produces an output field 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.

On input, EN editing is the same as F editing.

ES Editing

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.

On input, ES editing is the same as F editing.

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 edit descriptors can be processed between the edit descriptor for the real part and the edit descriptor for the imaginary part. Character string edit descriptors can be processed between the two edit descriptors on output only.

LOGICAL Editing (L)

The L*w* edit descriptor indicates that the field occupies *w* positions. The specified input/output list item must be of type LOGICAL.

The input field consists of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. 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. If a processor is capable of representing letters in both upper and lower case, a lower-case letter is equivalent to the corresponding upper-case letter in a LOGICAL input field.

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.

CHARACTER Editing (A)

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.

Generalized Editing (G)

The 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.

Generalized Integer Editing

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

Generalized Real and Complex Editing

The form and interpretation of the input field is the same as for F editing.

The method of representation in the output field depends on the magnitude of the data object being edited. If the decimal point falls just before, within, or just after the *d* significant digits to be printed, then the output is as for the F edit descriptor; otherwise, editing is as for the E edit descriptor.

Note that the scale factor *k* (see *"P Editing"* on page 30) has no effect unless the magnitude of the data object to be edited is outside the range that permits effective use of F editing.

Generalized Logical Editing

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

Generalized Character Editing

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

Control Edit Descriptors

Control edit descriptors affect format control or the conversions performed by subsequent data edit descriptors.

Position Editing (T, TL, TR, and X)

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.

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.

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.

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.

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 processor default (no pluses).

P Editing

The *k*P edit descriptor sets the value of the scale factor to *k*. The scale factor affects the Q, F, E, EN, ES, D, or G 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 E and 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.
- On output, with F editing, the scale factor effect is that the externally represented number equals the internally represented number times 10^k.

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.

Character String Edit Descriptors

The character string edit descriptors cause literal CHARACTER data to be output. They must not be used for input.

CHARACTER String Editing

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

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.

H Editing (obsolescent)

The *c*H edit descriptor causes character information to be written from the next *c* characters (including blanks) following the H of the *c*H edit descriptor in the list of format items itself. The *c* characters are called a *Hollerith constant.*

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
read (unit=1, fmt=*) i,j,k
```

all use list-directed formatting.

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

*r*c*

*r**

Where:

c is a literal constant 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 or slashes with optional preceding or following blanks; or one or more blanks between two non-blank values. A slash separator causes termination of the input statement after transfer of the previous value.

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

Table 4: List-Directed Input Editing

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, or begin with *r**.

List-Directed Output

For list-directed output the following formatting applies:

Table 5: List-Directed Output Editing

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

Namelist Formatting

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

1. optional blanks
2. the ampersand character followed immediately by the namelist group name specified in the namelist input/output statement
3. one or more blanks
4. a sequence of zero or more *name-value subsequences*, and
5. a slash indicating the end of the namelist record.

The characters in namelist records form a sequence of *name-value subsequences*. 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.

Example:

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

If the input records 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`.

Statements

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

Fortran statements can be grouped into five categories. They are

- Control Statements
- Specification Statements
- Input/Output Statements
- Assignment and Storage Statements
- Program Structure Statements

Control Statements

Arithmetic IF (obsolescent)

Execution of an arithmetic IF statement causes evaluation of an expression followed by a transfer of control. The branch target statement identified by the first, second, or third label in the arithmetic IF statement is executed next if the value of the expression is less than zero, equal to zero, or greater than zero, respectively.

Assigned GOTO (obsolescent)

The assigned GOTO 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. If the parenthesized list of labels is present, the variable must be one of the labels in the list.

CALL

The CALL statement invokes a subroutine and passes to it a list of arguments.

CASE

Execution of a SELECT CASE statement causes a 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.

Computed GOTO

The computed GOTO statement causes transfer of control to one of a list of labeled statements.

CONTINUE

Execution of a CONTINUE statement has no effect.

CYCLE

The CYCLE statement curtails the execution of a single iteration of a DO loop.

DO

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

ELSE IF

The ELSE IF statement controls conditional execution of a nested IF block in an IF construct where all previous IF expressions are false.

ELSE

The ELSE statement controls conditional execution of a block of code in an IF construct where all previous IF expressions are false.

ELSEWHERE

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.

END DO

The END DO statement ends a DO construct.

END FORALL

The END FORALL statement ends a FORALL construct.

END IF

The END IF statement ends an IF construct.

END SELECT

The END SELECT statement ends a CASE construct.

END WHERE

The END WHERE statement ends a WHERE construct.

ENTRY

The ENTRY statement permits one program unit to define multiple procedures, each with a different entry point.

EXIT

The EXIT statement terminates a DO loop.

FORALL

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

GOTO

The GOTO statement transfers control to a statement identified by a label.

IF

The IF statement controls whether or not a single executable statement is executed.

IF-THEN

The IF-THEN statement begins an IF construct.

PAUSE (Obsolescent)

The PAUSE statement temporarily suspends execution of the program.

RETURN

The RETURN statement completes execution of a subroutine or function and returns control to the statement following the procedure invocation.

SELECT CASE

The SELECT CASE statement begins a CASE construct. It contains an expression that, when evaluated, produces a case index. The case index is used in the CASE construct to determine which block in a CASE construct, if any, is executed.

STOP

The STOP statement terminates execution of the program.

WHERE

The WHERE statement is used to mask the assignment of values in array assignment statements. The WHERE statement can begin a WHERE construct that contains zero or more assignment statements, or can itself contain an assignment statement.

Specification Statements

ALLOCATABLE
The ALLOCATABLE statement declares allocatable arrays. The shape of an allocatable array is determined when space is allocated for it by an ALLOCATE statement.

CHARACTER
The CHARACTER statement declares entities of type CHARACTER.

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 an executable program.

COMPLEX
The COMPLEX statement declares names of type COMPLEX.

DATA
The DATA statement provides initial values for variables. It is not executable.

Derived-Type Definition Statement
The derived-type definition statement begins a derived-type definition.

DIMENSION
The DIMENSION statement specifies the shape of an array.

DLL_EXPORT (Windows only)
The DLL_EXPORT statement declares names externally available in a DLL.

DLL_IMPORT (Windows only)
The DLL_IMPORT statement declares names to import from a DLL.

DOUBLE PRECISION
The DOUBLE PRECISION statement declares names of type double precision REAL.

EQUIVALENCE
The EQUIVALENCE statement specifies that two or more objects in a scoping unit share the same storage.

EXTERNAL
The EXTERNAL statement specifies external procedures. Specifying a procedure name as EXTERNAL permits the name to be used as an actual argument.

IMPLICIT
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 specified in the statement. Alternately, it can specify that no implicit typing is to apply in the scoping unit.

INTEGER
The INTEGER statement declares names of type INTEGER.

INTENT
The INTENT statement specifies the intended use of a dummy argument.

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.

LOGICAL

The LOGICAL statement declares names of type LOGICAL.

NAMELIST

The NAMELIST statement specifies a list of variables which can be referred to by one name for the purpose of performing input/output.

ML_EXTERNAL (Windows only)

The ML_EXTERNAL statement specifies the name is an external procedure, or a common block name, that is available for calling from a mixed language procedure (see *"ML_EXTERNAL Statement"* on page 199).

MODULE PROCEDURE

The MODULE PROCEDURE statement specifies that the names in the statement are part of a generic interface.

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.

PARAMETER

The PARAMETER statement specifies named constants.

POINTER

The POINTER statement specifies a list of variables that have the POINTER attribute.

PRIVATE

The PRIVATE statement specifies that the names of entities are accessible only within the current module.

PUBLIC

The PUBLIC statement specifies that the names of entities are accessible anywhere the module in which the PUBLIC statement appears is used.

REAL

The REAL statement declares names of type REAL.

SAVE

The SAVE statement specifies that all objects in the statement retain their association, allocation, definition, and value after execution of a RETURN or subprogram END 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.

TARGET

The TARGET statement specifies a list of object names that have the target attribute and thus can have pointers associated with them.

TYPE

The TYPE statement specifies that all entities whose names are declared in the statement are of the derived type named in the statement.

USE

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.

VOLATILE

The VOLATILE statement specifies that a data object may be referenced, become redefined or undefined by means not specified in the Fortran standard (see *“VOLATILE Statement”* on page 267).

Input/Output Statements

BACKSPACE

The BACKSPACE statement positions the file before the current record, if there is a current record, otherwise before the preceding record.

CLOSE

The CLOSE statement terminates the connection of a specified input/output unit to an external file.

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.

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.

INQUIRE

The INQUIRE statement enables the program to make inquiries about a file’s existence, connection, access method or other properties.

OPEN

The OPEN statement connects or reconnects an external file and an input/output unit.

PRINT

The PRINT statement transfers values from an output list to an input/output unit.

READ

The READ statement transfers values from an input/output unit to the entities specified in an input list or a namelist group.

REWIND

The REWIND statement positions the specified file at its initial point.

WRITE

The WRITE statement transfers values to an input/output unit from the entities specified in an output list or a namelist group.

Assignment and Storage Statements

ALLOCATE

For an allocatable array the ALLOCATE statement defines the bounds of each dimension and allocates space for the array.

For a pointer the ALLOCATE statement creates an object that implicitly has the TARGET attribute and associates the pointer with that target.

ASSIGN (obsolescent)

Assigns a statement label to an INTEGER variable.

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.

DEALLOCATE

The DEALLOCATE statement deallocates allocatable arrays and pointer targets and disassociates pointers.

NULLIFY

The NULLIFY statement disassociates pointers.

Pointer Assignment

The pointer assignment statement associates a pointer with a target.

Program Structure Statements

BLOCK DATA

The BLOCK DATA statement begins a block data program unit.

CONTAINS

The CONTAINS statement separates the body of a main program, module, or subprogram from any internal or module subprograms it contains.

END

The END statement ends a program unit, module subprogram, interface, or internal subprogram.

FUNCTION

The FUNCTION statement begins a function subprogram, and specifies its return type and name (the function name by default), its dummy argument names, and whether it is recursive.

INTERFACE
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.

MODULE
The MODULE statement begins a module program unit.

PROGRAM
The PROGRAM statement specifies a name for the main program.

Statement Function
A statement function is a function defined by a single statement.

SUBROUTINE
The SUBROUTINE statement begins a subroutine subprogram and specifies its dummy argument names and whether it is recursive.

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;
- 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 6: Statement Order

PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement		
USE statements		
FORMAT and ENTRY statements	IMPLICIT NONE	
	PARAMETER statements	IMPLICIT statements
	PARAMETER and DATA statements	Derived-type definitions, interface blocks, type declaration statements, statement function statements, and specification statements
	DATA statements	Executable statements
CONTAINS statement		
Internal subprograms or module subprograms		
END statement		

Statements are restricted in what *scoping units* (see *“Scope”* on page 58) they may appear, as follows:

- An ENTRY statement may only appear in an external subprogram or module subprogram.
- A USE statement may not appear in a BLOCK DATA program unit.
- A FORMAT statement may not appear in a module scoping unit, BLOCK DATA program unit, or interface body.
- A DATA statement may not appear in an interface body.
- A derived-type definition may not appear in a BLOCK DATA program unit.
- An interface block may not appear in a BLOCK DATA program unit.
- A statement function may not appear in a module scoping unit, BLOCK DATA program unit, or interface body.
- An executable statement may not appear in a module scoping unit, a BLOCK DATA program unit, or an interface body.
- A CONTAINS statement may not appear in a BLOCK DATA program unit, an internal subprogram, or an interface body.

Executable Constructs

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

- The CASE and IF constructs control whether a block will be executed (see *“CASE Construct”* on page 88 and *“IF Construct”* on page 156).
- The DO construct controls how many times a block will be executed (see *“DO Construct”* on page 116).
- The FORALL construct controls multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements (see *“FORALL Construct”* on page 137).
- The WHERE construct controls which elements of an array will be affected by a block of assignment statements (see *“WHERE Construct”* on page 268).

Construct Names

Optional construct names can be used with CASE, IF, DO, and FORALL constructs. Use of construct names can add clarity to a program. For the DO construct, construct names enable a CYCLE or EXIT statement to leave a DO nesting level other than the current one. All construct names must match for a given construct. For example, if a SELECT CASE statement has a construct name, the corresponding CASE and END SELECT statements must have the same construct name.

Procedures

Fortran has two varieties of procedures: functions and subroutines. Procedures are further categorized in the following table:

Table 7: Procedures

Functions	Intrinsic Functions	Generic Intrinsic Functions
		Specific Intrinsic Functions
	External Functions	Generic External Functions
		Specific External Functions
	Internal Functions	
	Statement Functions	
Subroutines	Intrinsic Subroutines	Generic Intrinsic Subroutines
		Specific Intrinsic Subroutines
	External Subroutines	Generic External Subroutines
		Specific External Subroutines
	Internal Subroutines	

Intrinsic procedures are built-in procedures that are provided by the Fortran processor.

An *external procedure* is defined in a separate program unit and can be separately compiled. It is not necessarily coded in Fortran. External procedures and intrinsic procedures can be referenced anywhere in the program.

An *internal procedure* is contained within another program unit. It can only be referenced from within the containing program unit.

Internal and external procedures can be referenced recursively if the RECURSIVE keyword is included in the procedure definition.

Intrinsic and external procedures can be either *specific* or *generic*. A generic procedure has specific versions, which can be referenced by the generic name. The specific version used is determined by the type, kind, and rank of the arguments.

Additionally, procedures can be *elemental* or *non-elemental*. An elemental procedure can take as an argument either a scalar or an array. If the procedure takes an array as an argument, it operates on each element in the array as if it were a scalar.

Each of the various kinds of Fortran procedures is described in more detail below.

Intrinsic Procedures

Intrinsic procedures are built-in procedures provided by the Fortran processor. Fortran has over one hundred standard intrinsic procedures. Each is documented in detail in the Alphabetical Reference. A table is provided in *"Intrinsic Procedures"* on page 283.

Subroutines

A subroutine is a self-contained procedure that is invoked using a CALL statement. 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
  multiply = x*y
end subroutine multiply
```

This program calls the subroutine `multiply` and passes two REAL *actual arguments*, `a` and `b`. The subroutine `multiply`'s corresponding *dummy arguments*, `x` and `y`, refer to the same storage as `a` and `b` in `main`. When the subroutine returns, `a` has the value 48.0 and `b` is unchanged.

The syntax for a subroutine definition is

> *subroutine-stmt*
> *[use-stmts]*
> *[specification-part]*
> *[execution-part]*
> *[internal-subprogram-part]*
> *end-subroutine-stmt*

Where:
subroutine-stmt is a SUBROUTINE statement.

use-stmts is zero or more USE statements.

specification-part is zero or more specification statements.

execution part is zero or more executable statements.

internal-subprogram-part is
> CONTAINS
> *procedure-definitions*

procedure-definitions is one or more procedure definitions.

end-subroutine-stmt is
> END *[* SUBROUTINE *[subroutine-name]]*

subroutine-name is the name of the subroutine.

Functions

A function is a procedure that produces a single scalar or array result. It is used in an expression in the same way a variable is. 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, square
  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.

The syntax for a function reference is

function-name (*actual-arg-list*)

Where:
function-name is the name of the function.

actual-arg-list is a list of actual arguments.

A function can be defined as an internal or external function or as a statement function.

External Functions

External functions can be called from anywhere in the program. The syntax for an external function definition is

> *function-stmt*
> *[use-stmts]*
> *[specification-part]*
> *[execution-part]*
> *[internal-subprogram-part]*
> *end-function-stmt*

Where:

function-stmt is a FUNCTION statement.

use-stmts is zero or more USE statements.

specification-part is zero or more specification statements.

execution part is zero or more executable statements.

internal-subprogram-part is
> CONTAINS
> *procedure-definitions*

procedure-definitions is one or more procedure definitions.

end-function-stmt is
> END *[*FUNCTION *[function-name]]*

function-name is the name of the function.

Statement Functions

A statement function (see *"Statement Function"* on page 248) is a function defined on a single line with a single expression. It can only be referenced within the program unit in which it is defined. A statement function is best used where speed is more important than reusability in other locations, and where the function can be expressed in a single expression. The following is an example equivalent to the external function example in *"Functions"* on page 45:

```
program main
real :: a, b=3.6, c=3.8, square
square(x) = x*x
a = 3.7 + b + square(c) + sin(4.7)
print*, a
end
```

Internal Procedures

A procedure can contain other procedures, which can be referenced only from within the host procedure. Such procedures are known as *internal procedures*. An internal procedure is specified within the host procedure following a CONTAINS statement, which must appear after all the executable code of the containing subprogram. The form of an internal procedure is the same as that of an external procedure.

Example:

```
subroutine external ()
  ...
  call internal ()        ! reference to internal procedure
  ...

contains

  subroutine internal ()  ! only callable from external()
        ...
  end subroutine internal

end subroutine external
```

Names from the host procedure are accessible to the internal procedure. This is called *host association.*

Recursion

A Fortran procedure can reference itself, either directly or indirectly, only if the RECURSIVE keyword is specified in the procedure definition. A function that calls itself directly must use the RESULT option (see *“FUNCTION Statement”* on page 146).

Pure Procedures

Fortran procedures can be specified as PURE, meaning that there is no chance that the procedure would have any side effect on data outside the procedure. Only pure procedures can be used in specification expressions. The PURE keyword must be used in the procedure declaration.

Elemental Procedures

Fortran procedures can be elemental, meaning that they work on each element of an array argument as if the argument were a scalar. The ELEMENTAL keyword must be used in the procedure declaration. Note that all elemental procedures are also pure procedures.

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*.

Argument Intent

Because Fortran passes arguments by reference, unwanted side effects can occur when an actual argument's value is changed by the called procedure. To protect the program from such unwanted side effects, the INTENT attribute is provided. A dummy argument can have one of the following attributes:

- INTENT(IN), when it is to be used to input data to the procedure and not to return results to the calling subprogram;
- INTENT(OUT), when it is to be used to return results but not to input data; and
- INTENT(IN OUT), when it is to be used for inputting data and returning a result. This is the default argument intent.

The INTENT attribute is specified for dummy arguments using the INTENT statement or in a type declaration statement.

Keyword Arguments

Using keyword arguments, the programmer can specify explicitly which actual argument corresponds to which dummy argument, regardless of position in the actual argument list. To do so, specify the dummy argument name along with the actual argument, using the following syntax:

keyword = *actual-arg*

Where:

keyword is the dummy argument name.

actual-arg is the actual argument.

Example:

```
...
call zee(c=1, b=2, a=3)
...

subroutine zee(a,b,c)
...
```

In the example, the actual arguments are provided in reverse order.

A procedure reference can use keyword arguments for zero, some, or all of the actual arguments (see *"Optional Arguments"* below). For those arguments not having keywords, the order in the actual argument list determines the correspondence with the dummy argument list. Keyword arguments must appear after any non-keyword arguments.

Note that for a procedure invocation to use keyword arguments an explicit interface must be present (see *"Procedure Interfaces"* on page 51).

Optional Arguments

An actual argument need not be provided for a corresponding dummy argument with the OPTIONAL attribute. To make an argument optional, specify the OPTIONAL attribute for the dummy argument, either in a type declaration statement or with the OPTIONAL statement.

An optional argument at the end of a dummy argument list can simply be omitted from the corresponding actual argument list. Keyword arguments must be used 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 out) :: 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
call zee(b=3.0, c=8.5) ! a omitted, keywords necessary
```

It is usually necessary in a procedure body to know whether or not an optional argument has been provided. 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 must not be referenced except as an argument to the PRESENT function or as an optional argument in a procedure reference.

Note that for a procedure to have optional arguments an explicit interface must be present (see *"Procedure Interfaces"* on page 51). Many of the Fortran intrinsic procedures have optional arguments.

Alternate Returns (obsolescent)

A procedure can be made to return to a labeled statement in the calling subprogram using an *alternate return*. The syntax for an alternate return dummy argument is

   *

The syntax for an alternate return actual argument is

   * *label*

Where:
label is a labeled executable statement in the calling subprogram.

An argument to the RETURN statement is used in the called subprogram to indicate which alternate return in the dummy argument list to take. For example,

```
...
call zee(a,b,*200,c,*250)
...

subroutine zee(a, b, *, c, *)
  ...
  return 2        ! returns to label 250 in calling procedure
  ...
  return 1        ! returns to label 200 in calling procedure
  return          ! normal return
```

Dummy Procedures

A dummy argument can be the name of a procedure that is to be referenced in the called subprogram or is to appear in an interface block or in an EXTERNAL or INTRINSIC statement. The corresponding actual argument must not be the name of an internal procedure or statement function.

Procedure Interfaces

A procedure interface is all the characteristics of a procedure that are of interest to the Fortran processor when the procedure is invoked. These characteristics include the name of the procedure, the number, order, type parameters, shape, and intent of the arguments; whether the arguments are optional, and whether they are pointers; and, if the reference is to a function, the type, type parameters, and rank of the result, and whether it is a pointer. If the function result is not a pointer, its shape is an important characteristic. The interface can be explicit, in which case the Fortran processor has access to all characteristics of the procedure interface, or implicit, in which case the Fortran processor must make assumptions about the interface.

Explicit Interfaces

It is desirable, to avoid errors, to create explicit interfaces whenever possible. In each of the following cases, an explicit interface is mandatory:

If a reference to a procedure appears
- with a keyword argument,
- as a defined assignment,
- in an expression as a defined operator, or
- as a reference by its generic name;

or if the procedure has
- an optional dummy argument,
- an array-valued result,
- a dummy argument that is an assumed-shape array, a pointer, or a target,

- a CHARACTER result whose length type parameter value is neither assumed nor constant, or
- a result that is a pointer.

An interface is always explicit for intrinsic procedures, internal procedures, and module procedures. A statement function's interface is always implicit. In other cases, explicit interfaces can be established using an *interface block*:

Syntax:

interface-stmt
[interface-body] ...
[module procedure statement] ...
end-interface statement

Where:

interface-stmt is an INTERFACE statement.

interface-body is

function-stmt
[specification-part]
end stmt

or

subroutine-stmt
[specification-part]
end-stmt

module-procedure-stmt is a MODULE PROCEDURE statement.

end-interface-stmt is an END INTERFACE statement.

function-stmt is a FUNCTION statement.

subroutine-stmt is a SUBROUTINE statement.

specification-part is the specification part of the procedure.

end-stmt is an END statement.

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
    logical, intent(in) :: a, b
  end function y
end interface
```

In this example, explicit interfaces are provided for the procedures `x` and `y`. Any errors in referencing these procedures in the scoping unit of the interface block will be diagnosed at compile time.

Generic Interfaces

An INTERFACE statement with a *generic-name* (see *"INTERFACE Block"* on page 169) specifies a generic interface for each of the procedures in the interface block. In this way external generic procedures can be created, analogous to intrinsic generic procedures.

Example:

```
interface swap  ! generic swap routine
  subroutine real_swap(x, y)
    implicit none
    real, intent(in out) :: x, y
  end subroutine real_swap
  subroutine int_swap(x, y)
    implicit none
    integer, intent(in out) :: x, y
  end subroutine int_swap
end interface
```

Here the generic procedure `swap` can be used with both the REAL and INTEGER types.

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 (see *"INTERFACE Block"* on page 169).

A defined operation has the form

operator operand

for a defined unary operation, and

operand operator operand

for a defined binary operation, where *operator* is one of the intrinsic operators or a user-defined operator of the form

.operator-name.

where *.operator-name.* consists of one to 31 letters.

For example, either

```
a .intersection.  b
```

or

```
a * b
```

might be used to indicate the intersection of two sets. The generic interface block might look like

```
interface operator (.intersection.)
  function set_intersection (a, b)
    implicit none
    type (set), intent(in) :: a, b, set_intersection
  end function set_intersection
end interface
```

for the first example, and

```
interface operator (*)
  function set_intersection (a, b)
    implicit none
    type (set), intent(in) :: a, b, set intersection
  end function set_intersection
end interface
```

for the second example. The function `set_intersection` would then contain the code to determine the intersection of `a` and `b`.

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.

An intrinsic operation (such as addition) cannot be redefined for valid intrinsic operands. For example, it is illegal to redefine plus to mean minus for numeric types.

The functions specified in the interface block take either one argument, in the case of a defined unary operator, or two arguments, for a defined binary operator. The operand or operands in a defined operation become the arguments to a function specified in the interface block, depending on their type, kind, and rank. If a defined binary operation is performed, the left operand corresponds to the first argument and the right operand to the second argument. Both unary and binary defined operations for a particular operator may be specified in the same interface block.

Defined Assignment

The assignment operator may be extended using an interface block with INTERFACE ASSIGNMENT (see *"INTERFACE Block"* on page 169). The mechanism is similar to that used to resolve a defined binary operation (see *"Defined Operations"* on page 53), with the variable on the left side of the assignment corresponding to the first argument of a subroutine in the interface block and the data object on the right side corresponding to the second argument. The first argument must be INTENT(OUT) or INTENT(IN OUT); the second argument must be INTENT(IN).

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.

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
- Block Data Program Unit
- Module Program Unit

External Functions and Subroutines are described in *"Functions"* on page 45 and *"Intrinsic Procedures"* on page 44.

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-stmt]*
> *[use-stmts]*
> *[specification-part]*
> *[execution-part]*
> *[internal-subprogram-part]*
> *end-stmt*

Where:
program-stmt is a PROGRAM statement.

use-stmts is one or more USE statements.

specification-part is one or more specification statements or interface blocks.

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

internal-subprogram is one or more internal procedures.

end-stmt is an END statement.

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. A block data program unit may be referenced only in EXTERNAL statements in other program units.

The form of a block data program unit is

block-data-stmt
[specification-part]
end-stmt

Where:

block-data-stmt is a BLOCK DATA statement.

specification-part is one or more specification statements, other than ALLOCATABLE, INTENT, PUBLIC, PRIVATE, OPTIONAL, and SEQUENCE.

end-stmt is an END statement.

Module Program Units

Module program units provide a means of packaging anything that is required by more than one *scoping unit* (a scoping unit is a program unit, subprogram, derived type definition, or procedure interface body, excluding any scoping units it contains). Modules may contain type specifications, 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). Typical uses of modules include

- declaration and initialization of data to be used in more than one subprogram without using common blocks.
- specification of explicit interfaces for procedures.
- 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-stmt*
> *[use-stmts]*
> *[specification-part]*
> *[module-subprogram-part]*
> *end-stmt*

Where:
module-stmt is a MODULE statement.

use-stmts is one or more USE statements.

specification-part is one or more interface blocks or specification statements other than OPTIONAL or INTENT.

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

end-stmt is an END 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

    function swap_reals        !module procedure
      ...
    end function swap_reals
```

```
        function swap_integers  !module procedure
          ...
        end function swap_integers
    end module example
```

Module Procedures

Module procedures have the same rules and organization as external procedures. They are analogous to internal procedures, however, in that they have access to the data of the host module. 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.

Using Modules

Information contained in a module may be made available within another program unit via the USE statement. For example,

```
    use set_module
```

would give the current scoping unit access to the names in module `set_module`. If a name in `set_module` 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 set_module, 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 set_module, only : c, d
```

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

Forward references to modules are not allowed in LF95. If a module resides in a separate file from the code that uses the module, the module must be compiled before the code using the module. If a module and the code using the module are in the same source file, the compiler will compile the module in the proper order, regardless of where the module appears in the source file.

Scope

Names of program units, common blocks, and external procedures have global scope. That is, they may be referenced from anywhere in the program. A global name must not identify more than one global entity in a program.

Names of statement function dummy arguments have statement scope. The same name may be used for a different entity outside the statement, and the name must not identify more than one entity within the statement.

Names of implied-do variables in DATA statements and array constructors have a scope of the implied-do list. The same name may be used for a different entity outside the implied-DO list, and the name must not identify more than one entity within the implied-DO list.

Other names have local scope. The local scope, called a *scoping unit*, is one of the following:

- a derived-type definition, excluding the name of the derived type.
- an interface body, excluding any derived-type definitions or interface bodies within it.
- a program unit or subprogram, excluding derived-type component definitions, interface bodies, and subprograms contained within it.

Names in a scoping unit may be referenced from a scoping unit contained within it, except when the same name is declared in the inner, contained scoping unit. This is known as *host association*. For example,

```
subroutine external ()
  implicit none
  integer :: a, b
  ...

contains

  subroutine internal ()
    implicit none
    integer :: a
    ...
    a=b  ! a is the local a;
          ! 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.

Data Sharing

To make an entity available to more than one program unit, pass it as an argument, place it in a common block (see *“COMMON Statement”* on page 96), or declare it in a module and use the module (see *“Module Program Units”* on page 56).

2 Alphabetical Reference

This chapter contains descriptions and examples of Fortran 95 statements, constructs, intrinsic procedures, and extensions.

ABS Function

Description

The ABS function returns the absolute value of a numeric argument.

Syntax

ABS (*a*)

Arguments

a is an INTENT(IN) scalar or array of type REAL, INTEGER, or COMPLEX.

Result

If *a* is INTEGER or REAL, the result is the same type and kind as *a* and has the value $|a|$.

If *a* is COMPLEX with value (x,y), the result is a REAL value with the same kind as a, and is a representation of $\sqrt{x^2 + y^2}$.

Example

```
real :: y=-4.5
complex :: z=(1.,-1.)
write(*,*) abs(y)  ! writes 4.5000000
write(*,*) abs(z)  ! writes 1.4142135
```

ACHAR Function

Description
The ACHAR function returns a character from the ASCII collating sequence. See “ASCII Character Set” on page 319.

Syntax
ACHAR (*i*)

Arguments
i is an INTENT(IN) scalar or array of type INTEGER.

Result
A length one CHARACTER value corresponding to the character in position (*i*) of the ASCII collating sequence.

If *i* is an array, the result is an array of length one CHARACTER values, with the same shape as *i*

Example
```
integer, dimension(6) :: i=(/72,111,119,100,121,33/)
write(*,*) achar(i)  ! writes "Howdy!"
```

ACOS Function

Description
The ACOS function returns the trigonometric arccosine of a real number, in radians.

Syntax
ACOS (*x*)

Arguments
x is an INTENT(IN) scalar or array of type REAL and must be within the range $-1 \le x \le 1$. If the argument is outside this range, an error message is printed and program execution is terminated.

Result
A REAL representation, expressed in radians, of the arccosine of *x*.

Example
```
real :: x=.5
```

```
write(*,*) acos(x)  ! writes 1.0471975
```

ADJUSTL Function

Description

The ADJUSTL function results in a character string which has been adjusted to the left. Leading blanks are removed from any text and replaced as trailing blanks. The resulting character string is the same length as the input string.

Syntax

ADJUSTL (*string*)

Arguments

string is an INTENT(IN) scalar or array of type CHARACTER.

Result

A CHARACTER value the same length, kind, and shape as *string*.

The result is the text from *string* with any leading blanks removed and the same number of trailing blanks inserted.

Example

```
character(len=10) :: str="    string"
write(*,*) "’",adjustl(str),"’" ! writes ’string    ’
```

ADJUSTR Function

Description

The ADJUSTR function results in a character string which has been adjusted to the right. Trailing blanks are removed from any text string, and replaced as leading blanks. The resulting character string is the same length as the input string.

Syntax

ADJUSTR (*string*)

Arguments

string is an INTENT(IN) scalar or array of type CHARACTER.

Result

A CHARACTER of the same length, kind, and shape as *string*.

The result is the text from *string* with any trailing blanks removed and the same number of leading blanks inserted.

Example

```
character(len=10) :: str="string    "
write(*,*) "'",adjustr(str),"'" ! writes '    string'
```

AIMAG Function

Description

The AIMAG function returns the imaginary part of a complex number.

Syntax

AIMAG (*z*)

Arguments

z is an INTENT(IN) scalar or array of type COMPLEX.

Result

A REAL number with the same kind as *z*. If *z* has the value (*x*,*y*) then the result has the value *y*.

Example

```
complex :: z=(-4.2,5.5)
write(*,*) aimag(z)  ! writes 5.500000
```

AINT Function

Description

The AINT function truncates a real number by removing its fractional part.

Syntax

AINT (*a [, kind]*)

Required Arguments

a is an INTENT(IN) scalar or array of type REAL.

Optional Arguments

kind determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *"KIND Function"* or *"SELECTED_REAL_KIND Function"*.

Result

The result is equal to the value of *a* without its fractional part.

If *kind* is present, the result is a REAL value of kind *kind*, otherwise it is the same kind as *a*.

Example

```
real(kind=kind(1.e0)) :: r1=-7.32, r2=1.999999
real(kind=kind(1.d0)) :: dr
write(*,*) aint(r1,kind(r1)) ! writes -7.000000
write(*,*) aint(r2, kind(dr)) ! writes 1.0000000000000
```

ALL Function

Description

The ALL function determines whether all values in a logical mask are true either for an entire mask or along a given dimension of the mask.

Syntax

ALL (*mask [, dim]*)

Required Arguments

mask is an INTENT(IN) array of type LOGICAL. It cannot be scalar.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER with a value within the range $1 \leq x \leq n$, where *n* is the rank of *mask*. The corresponding actual argument cannot be an optional dummy argument.

Result

The result is of type LOGICAL and the same kind as MASK. Its value and rank are determined as follows:

1. The function will return a scalar logical value if *mask* has rank one, or if *dim* is absent. The result has the value true if all elements of *mask* are true.

2. The function will return a logical array of rank *n*-1 if *dim* is present and *mask* has rank two or greater. The resulting array is of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ 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 1

```
real, dimension(4) :: o=0.,p=1.,q=(/1.,-2.,3.,4./)
if (all(q /= 0.)) o=p/q
write(*,*) o ! writes 1.000000 -.5000000 .3333333 .2500000
```

Example 2

```
integer, dimension (2,3) :: a, b
a = reshape((/1,2,3,4,5,6/), (/2,3/))
write(*,'(2i3)') a ! writes 1 2
                   !        3 4
                   !        5 6
b = reshape((/1,2,3,5,6,4/), (/2,3/))
write(*,'(2i3)') b ! writes 1 2
                   !        3 4
                   !        5 6
write(*,*) all(a==b)   ! writes F
write(*,*) all(a==b, 1)! writes T F F
write(*,*) all(a==b, 2)! writes F F
```

ALLOCATABLE Statement

Description

The ALLOCATABLE statement declares arrays as having the allocatable attribute, and may also define the rank of an allocatable array. The shape of an allocatable array is determined when space is allocated for it by executing an ALLOCATE statement.

Syntax

ALLOCATABLE *[::] array-name [(deferred-shape)] [, array-name* (*deferred-shape*)*] ...*

Where:

array-name is the name of an array variable.

deferred-shape is : *[, :] ...* where the number of colons is equal to the rank of *array-name*.

Remarks

The Fortran 95 standard states that the object of the ALLOCATABLE statement must not be a dummy argument or function result. As an extension, dummy arrays are allowed to have the allocatable attribute.

If the DIMENSION of array-name is specified elsewhere in the scoping unit, it must be specified as a *deferred-shape*.

Example

```
integer :: a, c(:,:,:)
integer, allocatable :: b(:,:) ! allocatable attribute
allocatable a(:), c            ! allocatable statement
allocate (a(2),b(3,-1:1),c(10,10,10))! space allocated
write(*,*) shape(a),shape(b),shape(c)! writes 2 3 3 10 10 10
deallocate (a,b,c)                   ! space deallocated
```

ALLOCATE Statement

Description

The ALLOCATE statement dynamically creates storage for array variables having the ALLOCATABLE or POINTER attribute. If the object of an ALLOCATE statement is a pointer, execution of the ALLOCATE statement causes the pointer to become associated. If the object of an ALLOCATE statement is an array, the ALLOCATE statement defines the shape of the array.

Syntax

ALLOCATE (*allocation-list [*, STAT=*stat-variable]*)

Where:

allocation-list is a comma-separated list of pointer or allocatable variables. Each allocatable or pointer array in the *allocation-list* will have a list of dimension bounds, (*[lower-bound* :*]* *upper-bound [*, ...*]*)

upper bound and *lower-bound* are scalar INTEGER expressions.

stat-variable is a scalar INTEGER variable.

Remarks

When the ALLOCATE statement is executed, the number of dimensions being allocated must agree with the declared rank of the array.

If the optional STAT= is present and the ALLOCATE statement succeeds, *stat-variable* is assigned the value zero. If STAT= is present and the ALLOCATE statement fails, *stat-variable* is assigned the number of the error message generated at runtime.

If an error condition occurs during execution of an ALLOCATE statement that does not contain the STAT= specifier, execution of the program is terminated.

The default *lower-bound* value is one.

If *upper-bound* is less than *lower-bound*, the extent of that dimension is zero and the entire array has zero size.

The ALLOCATED intrinsic function determines whether an allocatable array is currently allocated.

The ASSOCIATED intrinsic function determines whether a pointer is currently associated with a target.

Attempting to allocate a currently allocated variable causes an error condition to occur.

If a pointer that is currently associated with a target is allocated, a new pointer target is created and the pointer is associated with that target. If there is no other reference to the original target, the storage associated with the original target is lost and cannot be recovered.

If local allocatable or pointer arrays do not have the SAVE attribute, they may be automatically deallocated upon execution of a RETURN statement.

Example 1

```
integer,pointer,dimension(:,:) :: i => null()
integer,allocatable,dimension (:) :: j
integer,pointer :: k                      ! scalar pointer
write(*,*) associated(i), associated(k) ! writes F F
allocate (i(10,20),k)
write(*,*) associated(i), shape(i)      ! writes T 10 20
write(*,*) associated(k)                ! writes T
deallocate (i,k)
write(*,*) allocated(j)                 ! writes F
allocate (j(10))
write(*,*) allocated(j), shape(j)       ! writes T 10
deallocate (j)                          ! space deallocated
```

Example 2

```
integer :: alloc_stat
real,allocatable,dimension (:) :: r
write(*,*) allocated(r)                ! writes F
allocate (r(10),stat=alloc_stat)
write(*,*) allocated(r),alloc_stat     ! writes T 0
allocate (r(20),stat=alloc_stat)
write(*,*) allocated(r),alloc_stat     ! writes T 1001
deallocate (r)                         ! space deallocated
allocate (r(20:-20),stat=alloc_stat)   ! zero size array
write(*,*) size(r),shape(r),alloc_stat ! writes 0 0 0
```

ALLOCATED Function

Description

The ALLOCATED function returns a true or false value indicating the status of an allocatable variable.

Syntax

ALLOCATED (*array*)

Arguments

array is an INTENT(IN) array with the allocatable attribute.

Result

The result is a scalar of default LOGICAL type. It has the value true if *array* is currently allocated and false if *array* is not currently allocated.

Example

```
integer, allocatable :: i(:)
write(*,*) allocated(i)  ! writes F
allocate (i(2))
write(*,*) allocated(i) ! writes T
```

ANINT Function

Description

The ANINT function rounds a REAL number up or down to the nearest whole number.

Syntax

ANINT (*a [, kind]*)

Required Arguments

a is an INTENT(IN) scalar or array of type REAL.

Optional Arguments

kind is INTENT(IN) and determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *“KIND Function”* or *“SELECTED_REAL_KIND Function”*.

Result

The result is a type REAL number representing the whole number nearest to the argument. If $a > 0$, the result has the value INT(a + 0.5); if $a \le 0$, the result has the value INT(a - 0.5).

If *kind* is present, the result is the same kind as *kind*.

If *kind* is absent, the result is the same kind as *a*.

Example

```
real :: x=7.73,y=1.5,z=-1.5
write(*,*) anint(x,kind(1.d0)) ! writes 8.000000000000000
write(*,*) anint(y)            ! writes 2.0000000
write(*,*) anint(z)            ! writes -2.0000000
```

ANY Function

Description:

The ANY function determines whether any values in a logical mask are true either for an entire mask or along a given dimension of the mask.

Syntax

ANY (*mask [, dim]*)

Required Arguments

mask is an INTENT(IN) array of type LOGICAL. It must not be scalar.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER with a value within the range $1 \le x \le n$, where *n* is the rank of *mask*. The corresponding actual argument cannot be an optional dummy argument.

Result

The result is of type LOGICAL and the same kind as MASK. Its value and rank are determined as follows:

1. The function will return a scalar logical value if *mask* has rank one, or if *dim* is absent. The result has the value true if any elements of *mask* are true.
2. The function will return a logical array of rank *n*-1 if *dim* is present and *mask* has rank two or greater. The resulting array is of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ 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 elements in that vector.

Example 1

```
real,dimension(4) :: q=(/1.,-2.,3.,4./)
write(*,*) any(q < 0.) ! writes T
```

Example 2

```
integer, dimension (2,3) :: a, b
a = reshape((/1,2,3,4,5,6/), (/2,3/))
write(*,'(2i3)') a ! writes 1 2
                   !        3 4
                   !        5 6
b = reshape((/1,2,3,5,6,4/), (/2,3/))
write(*,'(2i3)') b ! writes 1 2
                   !        3 4
                   !        5 6
write(*,*) any(a==b)    ! writes T
write(*,*) any(a==b, 1) ! writes T T F
write(*,*) any(a==b, 2) ! writes T T
```

Arithmetic IF Statement (obsolescent)

Description

Execution of an arithmetic IF statement causes evaluation of an expression followed by a transfer of control. The branch target statement identified by the first, second, or third label is executed if the value of the expression is less than zero, equal to zero, or greater than zero, respectively.

Syntax

IF (*expr*) *label*, *label*, *label*

Where:

expr is a scalar numeric expression.

label is a statement label.

Remarks

Each *label* must be the label of a branch target statement that appears in the same scoping unit as the arithmetic IF statement.

expr must not be of type COMPLEX.

The same *label* can appear more than once in one arithmetic IF statement.

The arithmetic IF statement is an ancient construct created in the early days of Fortran, and was suitable for the tiny programs which the machines of that era were able to execute. As hardware got better and programs grew larger, the arithmetic IF statement was identified as one of the main contributors to a logic snarled condition known as "spaghetti code", which made a program difficult to read and debug. The arithmetic IF statement was replaced by the *“IF Construct”*. While the arithmetic IF statement is obsolescent and should never be used when writing new code, it is fully supported.

Example

```
   real :: b=1.d0
10 write(*,*) " arithmetic if construct"
   if (b) 20,10,30
20 write(*,*) " if b < 0, control is transferred here"
30 write(*,*) " if b > 0, control is transferred here"
   write(*,*) " equivalent if construct"
   if( b < 0. ) then
     write(*,*) "if b < 0, control is transferred here"
   else if ( b > 0. ) then
     write(*,*) " if b > 0, control is transferred here"
   else
     write(*,*) " if b=0, control is transferred here"
   end if
```

ASIN Function

Description

The ASIN function returns the trigonometric arcsine of a real number, in radians.

Syntax

ASIN (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL and must be within the range $-1 \le x \le 1$. If the argument is outside this range, an error message is printed and program execution is terminated.

Result

A REAL representation, expressed in radians, of the arcsine of *x*.

Example

```
real :: x=.5
write(*,*) asin(x)  ! writes .523599
```

Assigned GOTO Statement (obsolescent)

Description

The assigned GOTO 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. If the parenthesized list of labels is present, the variable must be one of the labels in the list.

Syntax

GOTO *assign-variable [[,]* (*labels*)*]*

Where:

assign-variable is a scalar INTEGER variable that was assigned a label in an ASSIGN statement.

labels is a comma-separated list of statement labels.

Remarks

At the time of execution of the GOTO statement, *assign-variable* must be defined with the value of a label of a branch target statement in the same scoping unit.

The assigned GOTO statement is a construct created in the early days of Fortran, and was suitable for the tiny programs which the machines of that era were able to execute. As hardware got better and programs grew larger, the assigned GOTO statement was identified as a major contributor to a logic snarled condition known as "spaghetti code", which made a program difficult to read and debug. The assigned GOTO statement may be replaced by the *"CASE Construct"* or the *"IF Construct"*. Although the assigned GOTO statement is obsolescent and should never be used when writing new code, it is fully supported.

Example

```
   assign 10 to i
   goto i
20 assign 30 to i
   goto i
10 write(*,*) " assigned goto construct"
   assign 20 to i
   goto i, (10,20,30)
30 continue
```

ASSIGN Statement (obsolescent)

Description

The ASSIGN statement assigns a statement branch label to an INTEGER variable.

Syntax

ASSIGN *label* TO *assign-variable*

Where:

label is a statement label.

assign-variable is a scalar INTEGER variable.

Remarks

assign-variable must be a named variable of default INTEGER kind. It must not be a structure component or an array element.

label must be the target of a branch target statement or the label of a FORMAT statement in the same scoping unit.

When defined with an INTEGER value, *assign-variable* must not be used as a label.

When assigned a label, *assign-variable* must not be used as anything except a label.

The ASSIGN statement is an ancient construct that induces the coder to write "spaghetti code", leading to difficult to read and debug programs. The assigned GOTO construct has been replaced by the structured *"CASE Construct"* or *"IF Construct"*. The ASSIGN statement is obsolescent and should never be used when writing new code. However, it is fully supported by the compiler.

Example

```
      assign 100 to i
100   continue
      goto i
```

Assignment Statement

Description

The assignment statement assigns the value of the expression on the right side of an equal sign to the variable on the left side of the equal sign.

Syntax

variable=expression

Where:

variable is a scalar variable, an array, or a variable of derived type.

expression is an expression whose result is conformable with *variable.*

Remarks

A numeric variable can only be assigned a numeric value; a CHARACTER variable can only be assigned a CHARACTER value of the same kind; a LOGICAL variable can only be assigned a LOGICAL value; a derived type variable can only be assigned a value of the same derived type.

Evaluation of *expression* takes place before the assignment is made.

If the kind of *expression* is different from that of *variable*, the result of *expression* undergoes an implicit type conversion to the kind and type of *variable*, possibly causing a loss of precision.

If *expression* is an array, then *variable* must be an array. If *expression* is scalar and *variable* is an array, all elements of *variable* are assigned the value of *expression*.

If variable is a pointer, it must be associated with a target. The target is assigned the value of *expression.*

If *variable* and *expression* are of CHARACTER type with different lengths, *expression* is truncated if longer than *variable*, and padded on the right with blanks if *expression* is shorter than *variable*.

Example 1

```
! Basic assignment examples
integer :: i1,i2
real :: r1,r2
real(kind(1.d0)) :: d1,d2
complex :: q1,q2
logical :: l1,l2
character(len=6) :: c1,c2
! assignment to a constant
i1=12345
r1=12345.
d1=12345.d0
q1=cmplx(1.2345e0,-1.2345e0)
l1=.true.
c1="Howdy!"
write(*,*) i1,r1,d1,q1,l1,c1
! assignment to an expression
i2=i1/10
r2=r1*10.
d2=d1**10.d0
q2=q1/r1
l2=(r1 < d1)
c2=c1(1:1) // "ow" // c2(4:6)
write(*,*) i2,r2,d2,q2,l2,c2
```

Example 2

```
! Conversion examples
integer(kind=selected_int_kind(4)) :: ishort=12345
integer(kind=selected_int_kind(6)) :: ilong
real(kind(1.e0)) :: a=1.234567e6, b1, b2
real(kind(1.d0)) :: d1, d2
! safe conversions
ilong=ishort
b1=ishort
b2=ilong
d1=ishort
write(*,*) ishort,ilong,b1,b2,d1
! dangerous conversions
ilong=a      ! succeeds this time
ishort=ilong ! overflows
write(*,*) a,ishort,ilong
ishort=a     ! overflows
write(*,*) ishort,a
ilong=a*b1 ! overflows
write(*,*) ilong,a*b1
! loss of precision
d1=exp(1.5d0)  ! no loss of precision
b1=d1          ! loses precision
d2=b1          ! d2 given a single precision value
write(*,'(4(/,g21.14))') exp(1.5d0),d1,d2,b1
b2=huge(ilong) ! loses precision
write(*,*) b2,huge(ilong)
```

Example 3

```
! array and derived type assignments
   real :: a=0.
   real,dimension(3) :: a_a1=0.
   real,dimension(3,3) :: a_a2=0.
   type realtype1                  ! derived type definition
     real :: a=0.
     real,dimension(3) :: a_a1=0.
     real,dimension(3,3) :: a_a2=0.
   end type realtype1
   type realtype2                  ! derived type definition
     real :: a=0.
     real,dimension(3) :: a_a1=0.
     real,dimension(3,3) :: a_a2=0.
   end type realtype2
   type (realtype1) :: rt1         ! derived type declaration
   type (realtype2) :: rt2         ! derived type declaration
! array assignment to a scalar constant
   a_a1=13.
   a_a2=16.
   rt1%a_a2=19.
   write(*,10) a_a1, a_a2, rt1%a_a2
! Array assignment to a scalar variable
   a_a1=a
   a_a2=a
   rt1%a_a2=a
   write(*,10) a_a1,a_a2,rt1%a_a2
! Array assignment to an array constant
   a_a1=(/1.,2.,3./)
   a_a2=reshape((/1.,2.,3.,4.,5.,6.,7.,8.,9./),(/3,3/))
   rt1%a_a2=reshape((/9.,8.,7.,6.,5.,4.,3.,2.,1./),(/3,3/))
   write(*,10) a_a1,a_a2,rt1%a_a2
! Array assignment to a derived type constant
   rt2=realtype2 (0.,(/1.,2.,3./),&
       reshape((/1.,2.,3.,4.,5.,6.,7.,8.,9./),(/3,3/)))
   write(*,20) rt2%a, &
         rt2%a_a1,rt2%a_a2
! Conformable assignments
   rt2%a=sum(a_a1)
   rt1%a_a1=a_a1
   rt2%a_a1=rt1%a_a1
   rt2%a_a2(3,:)=a_a1
   rt2%a_a2(1:2,1:2)=rt1%a_a2(1:2,2:3)
   write(*,20) rt2%a, rt2%a_a1, rt2%a_a2
10 format(/,3f7.3,2(/,3(/,3f7.3)),/)
20 format(/,f7.3,//,3f7.3,/,3(/,3f7.3),/)
! nonconformable assignments
! will produce error messages
!    a=a_a2
!    rt1=0.
!    rt1=a
!    rt1=rt2
!    rt1%a_a1=a_a2
!    rt1%a_a2=a_a1
```

ASSOCIATED Function

Description

The ASSOCIATED function indicates whether a pointer is associated or disassociated. It may also test a pointer for association with a particular target.

Syntax

ASSOCIATED (*pointer[*, *target]*)

Required Arguments

pointer is an INTENT(IN) variable with the pointer attribute whose association status is either associated or disassociated. The association status of *pointer* must not be undefined.

Optional Arguments

target is INTENT(IN) and must have either the pointer or target attribute. If it is a pointer, its pointer association status must not be undefined.

Result

The result is of type default LOGICAL.

When *target* is absent, the result is true if *pointer* is currently associated with a target. If *pointer* is disassociated, the result is false.

When *target* is present, the result is true if *pointer* is currently associated with *target.* The result is false if *pointer* is disassociated or associated with a different target.

If *target* has the pointer attribute, the result is true if both *pointer* and *target* are currently associated with the same target. If either *pointer* or *target* is disassociated, or if they are associated with different targets, the result is false.

Example

```
real,pointer :: a(:)
real,allocatable, target :: b(:)
write (*,*) associated(a) ! pointer disassociated by default
allocate(a(4))            ! a is associated
write (*,*) associated(a)
deallocate(a)             ! a is disassociated
write (*,*) associated(a)
allocate(b(5))
a => b                    ! a is associated with b
write (*,*) associated(a,b)
deallocate(b)             ! careful, a is undefined!!!
a => null()               ! a is disassociated
write(*,*) associated(a)
```

ATAN Function

Description

The ATAN function returns the arctangent of a real number, in radians.

Syntax

ATAN (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is a REAL representation of the arctangent of *x*, expressed in radians. The result always falls within the range $-\pi/2 \le x \le \pi/2$.

Example

```
real :: a=0.5
write(*,*) atan(a)  ! writes 0.4636476
```

ATAN2 Function

Description

The ATAN2 function returns the arctangent of *y*/*x*, expressed as radians. The ATAN2 function is able to return a greater range of values than the ATAN function, because it considers the sign of *x* and *y*.

Syntax

ATAN2 (*y*, *x*)

Arguments

y is an INTENT(IN) scalar or array of type REAL.

x is INTENT(IN) and of the same kind as *y*. If *y* is an array, x must be an array conformable to y. If *y i*s zero, *x* cannot be zero.

Result

The result is of the same kind as *y*. Its value is a REAL representation, expressed in radians, of the principal value of the argument of the complex number *x* + i*y*. The result falls within the range $-\pi < x \le \pi$.

If *y* is positive, the result is positive. If *y* is negative, the result is negative. If *y* is zero and *x* > 0, the result is zero. If *y* is zero and *x* < 0, the result is pi. If *x* is zero, the result is a representation of the value $\pi/2$ having the sign of *y*.

Example

```
real :: y=1.,x=1.
write(*,*) atan2(y,x)   ! writes 0.78539818
write(*,*) atan2(-y,x)  ! writes -0.78539818
write(*,*) atan2(y,-x)  ! writes 2.3561945
write(*,*) atan2(-y,-x) ! writes -2.3561945
write(*,*) atan2(0.,x)  ! writes 0.0000000
write(*,*) atan2(0.,-x) ! writes 3.1415927
write(*,*) atan2(y,0.)  ! writes 1.5707963
write(*,*) atan2(-y,0.) ! writes -1.5707963
```

BACKSPACE Statement

Description

The BACKSPACE statement moves the position of a file opened for sequential access to the beginning of the current record. If there is no current record, the file position is moved to the beginning of the preceding record. If there is no preceding record, the file position is unchanged.

Syntax

BACKSPACE *unit-number*

or

BACKSPACE (*position-spec-list*)

Where:

unit-number is a scalar INTEGER expression corresponding to the input/output unit number of an external file.

position-spec-list is *[[*UNIT *=] unit-number][*, ERR=*label][*, IOSTAT=*stat]* where UNIT=, ERR=, and IOSTAT= can be in any order but if UNIT= is omitted, then *unit-number* must be first.

label is a statement label that is branched to if an error condition occurs during execution of the statement.

stat is a variable of type INTEGER that is assigned a positive value if an error condition occurs, a negative value if an end-of-file or end-of-record condition occurs, and zero otherwise.

Remarks

If there is no current record and no preceding record, the file position is left unchanged.

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.

Backspacing a file that is connected but does not exist is prohibited.

Backspacing over records using list-directed or namelist formatting is prohibited.

Note that BACKSPACE may only be used on sequential access files.

Example

```
integer :: ios
backspace 10             ! backspace file on unit 10
backspace(10,iostat=ios) ! backspace with status return
```

BIT_SIZE Function

Description

The BIT_SIZE function returns the number of bits in a data object of type INTEGER.

Syntax

BIT_SIZE (*i*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

Result

The result is the same kind as *i*. Its value is equal to the number of bits in an integer of kind *i*.

Example

```
integer(kind=selected_int_kind(4)) :: i
integer(kind=selected_int_kind(12)) :: j
integer,dimension(2) :: k
write(*,*) bit_size(i) ! writes 16
write(*,*) bit_size(j) ! writes 64
write(*,*) bit_size(k) ! writes 32
```

BLOCK DATA Statement

Description

The BLOCK DATA statement begins a block data program unit. The block data program unit initializes data that appears in named common blocks.

Syntax

BLOCK DATA *[block-data-name]*

Where:

block-data-name is an optional name given to the block data program unit.

Remarks

There can only be one unnamed BLOCK DATA program unit in a program.

A block data program unit may only initialize variables that appear in a named common block.

The same named common block may not appear in more than one block data subprogram.

A block data subprogram may only contain type declaration statements; the attribute specifiers PARAMETER, DIMENSION, POINTER, SAVE and TARGET; the specification statements USE, IMPLICIT, COMMON, DATA, EQUIVALENCE and INTRINSIC.

A type declaration statement in a block data subprogram may not specify the attributes ALLOCATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE or PUBLIC.

Example

```
block data mydata
  common /d/ a,b,c
  data a,b,c /1.0,2.0,3.0/
end block data mydata
```

BTEST Function

Description

The BTEST function will test the bit in position *pos* in an INTEGER data object.

Syntax

BTEST (*i*, *pos*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER to be tested.

pos is an INTENT(IN) scalar or array of type INTEGER. It must be non-negative and less than BIT_SIZE (*i*). Bits are numbered from least significant to most significant, beginning with 0.

If both *i* and *pos* are arrays, they must be conformable.

Result

The result is of type default LOGICAL.

If both *i* and *pos* are scalar, BTEST returns the value true if bit *pos* has the value 1 and false if bit *pos* has the value zero.

If *i* is scalar and *pos* is an array, the result is a LOGICAL array the same shape as *pos*. Each element of the array contains the result of testing *i* for each bit position contained in each element of *pos*.

If *i* is an array and *pos* is scalar, the result is a LOGICAL array the same shape as *i*. Each element of the array contains the result of testing bit position *pos* for each element of array *i*.

If *i* and *pos* are conformable arrays, The result is a LOGICAL array the same shape as *i*. Each element of the array contains the result of testing each element of *i* using the bit position from the corresponding element of *pos*.

Example

```
integer :: i=-1,j=4,spos=bit_size(i)-1
integer :: k(4)=(/1,2,4,8/),pos(4)=(/0,1,2,3/)
write(*,*) btest(i,spos),btest(j,spos) ! test sign bit
write(*,*) btest(i,pos) ! test first 4 bits of i
write(*,*) btest(k,2)   ! test bit #2 for each element of k
write(*,*) btest(k,pos) ! test each element of k using the
                        ! corresponding element of pos
```

CALL Statement

Description

The CALL statement invokes a subroutine and passes it an arguments list.

Syntax

CALL *subroutine-name [([actual-arg-list])]*

Where:

subroutine-name is the name of a subroutine.

actual-arg-list is *[[keyword =] actual-arg] [, ...]*

keyword is the name of a dummy argument to *subroutine-name*.

actual-arg is an expression, a variable, a procedure name, or an *alternate-return-spec.*

*alternate-return-spec is *label*

label is a statement label.

Remarks

General:

actual-arg-list defines the correspondence between the *actual-arg*s supplied and the dummy arguments of the subroutine.

If *keyword*= is present, the actual argument is passed to the dummy argument whose name is the same as *keyword*. If a *keyword*= is absent, the actual argument is passed to the dummy argument in the corresponding position in the dummy argument list.

keyword= must appear with an *actual-arg* unless no previous *keyword*= has appeared in the *actual-arg-list.*

keyword= can only appear if the interface of the procedure is explicit in the scoping unit.

An *actual-arg* can be omitted if the corresponding dummy argument has the OPTIONAL attribute. Each *actual-arg* must be associated with a corresponding dummy argument.

Data objects as arguments:

An actual argument must be of the same kind as the corresponding dummy argument.

If the dummy argument is an assumed-shape array of type default CHARACTER, its length must agree with that of the corresponding actual argument.

The total length of a dummy argument of type default CHARACTER must be less than or equal to that of the corresponding actual argument.

If the dummy argument is a pointer, the actual argument must also be a pointer of the same type, attributes, and rank. At the invocation of the subroutine, the dummy argument pointer receives the pointer association status of the actual argument. At the end of the subroutine, the actual argument receives the pointer association status of the dummy argument.

If the actual argument has the TARGET attribute, any pointers associated with it remain associated with the actual argument. If the dummy argument has the TARGET attribute, any pointers associated with it become undefined when the subroutine completes.

The ranks of dummy arguments and corresponding actual arguments must agree unless the actual argument is an element of an array that is not an assumed-shape or pointer array, or a substring of such an element.

If an actual argument has the INTENT(OUT) attribute, and its value is not set within the subroutine, upon return from the subroutine, its value will be undefined.

Procedures as arguments:

If a dummy argument is a dummy procedure, the associated actual argument must be the specific name of an external, module, dummy, or intrinsic procedure.

The intrinsic functions AMAX0, AMAX1, AMIN0, AMIN1, CHAR, DMAX1, DMIN1, FLOAT, ICHAR, IDINT, IFIX, INT, LGE, LGT, LLE, LLT, MAX0, MAX1, MIN0, MIN1, REAL, and SNGL may not be associated with a dummy procedure. The results of these functions may be used as actual arguments.

If a generic intrinsic function name is also a specific name, only the specific procedure is associated with the dummy argument.

If a dummy procedure has an implicit interface either the name of the dummy argument is explicitly typed or the procedure is referenced as a function. The dummy procedure must not be called as a subroutine and the actual argument must be a function or dummy procedure.

If a dummy procedure has an implicit interface and the procedure is called as a subroutine, the actual argument must be a subroutine or a dummy procedure.

Alternate returns as arguments:
If a dummy argument is an asterisk, the corresponding actual argument must be an *alternate-return-spec*. The *label* in the *alternate-return-spec* must identify an executable construct in the scoping unit containing the procedure reference.

Example 1

```
! basic calling syntax
  real :: x=1.,y=2.
  call alpha(x,y)
end program
subroutine alpha(a,b)
  real :: a,b
  write(*,*) a,b
end subroutine alpha
```

Example 2

```
! calling with optional arguments
  interface
    subroutine alpha(a,b) ! define keywords here
      real :: a  ! use of optional arguments
      real,optional :: b  ! requires an interface
    end subroutine
  end interface
  real :: x=1., y=2.
  call alpha(x)        ! call with no options
  call alpha(x,y)      ! positional call
  call alpha(b=x, a=y) ! keyword call
end program
subroutine alpha (a,b)
  real :: a
  real,optional :: b
  if(present(b)) then ! b must not appear unless it
    write(*,*) a,b    !   is inside a construct that
  else                !   tests for its presence
    write(*,*) a
  end if
end subroutine alpha
```

Example 3

```
! calling with a procedure argument
  real,external :: euler
  call alpha(euler)
end program
subroutine alpha(f)
  real,external :: f
  write(*,*) f()
end subroutine alpha
real function euler()
  euler=exp(cmplx(0.,atan2(0.,-1.)))
end function euler
```

Example 4

```
! calling with the intent attribute
  real :: x=1.,y=2.
  call alpha(x,y)
  write(*,*) x,y
end program
subroutine alpha(a,b)
  real,intent(in) :: a  ! a cannot be changed inside alpha
  real,intent(out) :: b ! b must be initialized before
  b=a                   ! the dummy argument or actual
end subroutine alpha    ! argument is referenced
```

CARG Function

Description

The CARG function passes a numeric or logical argument by value, rather than using the Fortran standard of passing arguments by reference. If the argument is of type CHARACTER, the CARG function will convert the argument to a C string. CARG can only be used as an actual argument when invoking a subroutine or function.

Syntax

CARG (*item*)

Arguments

item is an INTENT(IN) named data object of any intrinsic type except COMPLEX and four-byte LOGICAL. It is the data object for which to return a value.

Result

If the argument is numeric or logical, the value of *item* is placed on the calling stack, rather than its address.

If the argument is of type CHARACTER, the Fortran length descriptor is removed and the character string is null terminated.

The C data type of the result is shown in Table 8.

Table 8: CARG result types

Fortran Type	Fortran Kind	C type
INTEGER	1	signed char
INTEGER	2	signed short int
INTEGER	4	signed long int
REAL	4	float

COMPLEX	4	must not be passed by value; if passed by reference (without CARG) it is a pointer to a structure of the form:
		
		struct complex {
		float real_part;
		float imaginary_part;};

LOGICAL	1	unsigned char
LOGICAL	4	must not be passed by value or by reference
CHARACTER	1	char *

Example

```
real :: a=1.0
character :: c="howdy"
i=my_c_function(carg(a))      ! a is passed by value
call my_c_subroutine(carg(c)) ! c is passed as a C string
```

CASE Construct

Description

The CASE construct selects blocks of executable code based on the value of an expression. A default case may be provided.

The SELECT CASE statement signals the beginning of a CASE construct. It contains an expression that, when evaluated, produces a case index. The case index in the CASE construct determines which block in a CASE construct, if any, is executed.

The CASE statement defines a case selector which, when matched with the value from a SELECT CASE statement, causes the following block of code to be executed.

The END SELECT statement signals the end of the innermost nested CASE construct.

Syntax

```
[construct-name :] SELECT CASE (case-expr)
CASE (case-selector [, case-selector] ...  ) [construct-name]
        block
        ...
[CASE DEFAULT [construct-name]]
        block
        ...
END SELECT [construct-name]
```

Where:

construct-name is an optional name for the CASE construct

case-expr is a scalar expression of type INTEGER, LOGICAL, or CHARACTER

case-selector is *case-value*
or : *case-value*
or *case-value* :
or *case-value* : *case-value*

case-value is a constant scalar LOGICAL, INTEGER, or CHARACTER expression.

block is a sequence of zero or more statements or executable constructs.

Remarks

Execution of a SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the case index.

Execution of a CASE code block occurs if the case index derived from the SELECT CASE statement is in the range specified by the CASE statement's case-selector. Execution of the code block ends when any subsequent CASE or END CASE statement is encountered, and the innermost case construct is exited.

The case-selector is evaluated as follows:

case-value means equal to *case-value*;

:*case-value* means less than or equal to *case-value*;

case-value: means greater than or equal to *case-value*; and

case-value:*case-value* means greater than or equal to the left *case-value*,

and less than or equal to the right *case-value.*

Each *case-value* must be of the same type and kind as the case construct's case index.

The ranges of *case-value*s in a case construct must not overlap.

If *case-value* is of type LOGICAL, it cannot have a range.

The block following a CASE DEFAULT, if any, is executed if the case index matches none of the *case-value*s in the case construct. CASE DEFAULT can appear before, among, or after other CASE statements, or can be omitted.

The *case-value*s in a case construct must not overlap.

Only one CASE DEFAULT is allowed in a given case construct.

If the SELECT CASE statement is identified by a *construct-name*, the corresponding END SELECT statement must be identified by the same construct name. If the SELECT CASE statement is not identified by a *construct-name*, the corresponding END SELECT statement must not have a *construct-name*.

Example 1

```
integer :: i=3
select case (i)
case (:-2)
  write(*,*) "i is less than or equal to -2"
case (0)
  write(*,*) "i is equal to 0"
case (1:97)
  write(*,*) "i is in the range 1 to 97, inclusive"
case default
  write(*,*) "i is either -1 or greater than 97"
end select
```

Example 2

```
character(len=5) :: c="Howdy"
select case (c)
case ("Hi","Hello","Howdy")
  write(*,*) "Hello, how are you?"
case ("Good Morning")
  write(*,*) "Nice morning, isn't it?"
case ("Good Night")
  write(*,*) "Goodbye."
case default
  write(*,*) "What time is it?"
end select
```

CEILING Function

Description

The CEILING function returns the smallest INTEGER number greater than or equal to a REAL number.

Syntax

CEILING (*a [, kind]*)

Required Arguments

a is an INTENT(IN) scalar or array of type REAL.

Optional Arguments

kind is INTENT(IN) and determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *"KIND Function"* or *"SELECTED_INT_KIND Function"*.

Result

The result is an INTEGER number whose value is the smallest integer greater than or equal to *a*.

If *kind* is present, it specifies the kind of the result.

If *kind* is absent, the result is type default INTEGER.

Example

```
real :: r=4.7, x(3)=(/-.5,0.,.5/)
write(*,*) ceiling(r)                      ! writes 5
write(*,*) ceiling(-r,selected_int_kind(2)) ! writes -4
write(*,*) ceiling(x)                      ! writes 0 0 1
```

CHAR Function

Description

The CHAR function returns a character from a specified character set.

Syntax

CHAR (*i [, kind]*)

Required Arguments

i is an INTENT(IN) scalar or array of type INTEGER. Each *i* value must be positive and not greater than the number of characters in the collating sequence of the character set specified by *kind*.

Optional Arguments

kind is INTENT(IN) and determines which character set is chosen. It must be a scalar INTEGER expression that can be evaluated at compile time. Only the ASCII character set is supported, with a kind number of 1. See “ASCII Character Set” on page 319.

Result

The result is a CHARACTER value of length one corresponding to the *i*th character of the given character set.

If *kind* is present, the resulting kind is specified by *kind*. Only the default kind is supported.

If *kind* is absent, the kind is of type default CHARACTER.

Example

```
integer,dimension(6) :: i=(/72,111,119,100,121,33/)
write(*,*) char(i)  ! writes "Howdy!"
```

CHARACTER Statement

Description

The CHARACTER statement declares entities having the CHARACTER data type.

Syntax

CHARACTER *[char-selector] [, attribute-list* ::*] entity [, entity] ...*

Where:
char-selector is:
char-selector is (*[*LEN=*] length [*, KIND=*kind])*
or (KIND=*kind [*, LEN=*length]*)
or * *char-length [,]*

kind is a scalar INTEGER expression that can be evaluated at compile time.

length is a scalar INTEGER expression that can be evaluated on entry to the program unit.
or *

char-length is a scalar INTEGER literal constant
or (*)

attribute-list is a comma-separated list from the following attributes: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), EXTERNAL, INTENT(IN) or INTENT(OUT) or INTENT(IN OUT), PUBLIC or PRIVATE, INTRINSIC, OPTIONAL, POINTER, SAVE, TARGET, VOLATILE, DLL_EXPORT or DLL_IMPORT or ML_EXTERNAL.

entity is:
entity-name [(*array-spec*)*] [* char-length] [=initialization-expr]*
or *function-name [*(*array-spec*)*] [* char-length]*

array-spec is an array specification

initialization-expr is a CHARACTER-valued expression that can be evaluated at compile time

entity-name is the name of the entity being declared

function-name is the name of a function being declared

Remarks

If *char-length* is not specified, the length is one.

An asterisk can be used for *char-length* only in the following ways:

1. If the entity is a dummy argument. The dummy argument assumes the length of the associated actual argument.

2. To declare a named constant. The length is that of the constant value.

3. In an external function, as the length of the function result. In this case, the function name must be declared in the calling scoping unit with a length other than *, or access such a definition by host or use association. The length of the result variable is assumed from this definition.

char-length for CHARACTER-valued statement functions and statement function dummy arguments must be a constant INTEGER expression.

The optional comma following * *char-length* in a *char-selector* is permitted only if no double colon appears in the statement.

The value of *kind* must specify a character set that is valid for this compiler.

char-length must not include a kind parameter.

The * *char-length* in *entity* specifies the length of a single entity and overrides the length specified in *char-selector*.

function-name must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

The =*initialization-expr* must appear if the statement contains a PARAMETER attribute.

If =*initialization-expr* appears, a double colon must appear before the list of *entities*. Each *entity* has the SAVE attribute, unless it is in a named common block.

The =*initialization-expr* must not appear if *entity-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 pointer, an external name, an intrinsic name, or an automatic object.

If an array or function with an array result is declared with the POINTER attribute, it must have a deferred shape.

If an array is declared with the ALLOCATABLE attribute, it must have a deferred shape.

If an array or function with an array result does not have the POINTER or the ALLOCATABLE attribute, it must be specified with an explicit shape.

If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attributes must not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attributes must not be specified.

The PARAMETER attribute cannot be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

The INTENT(IN), INTENT(OUT), INTENT(IN OUT), and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* may not have the PUBLIC attribute if its type has the PRIVATE attribute.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

An *entity* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

A subprogram must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

An *entity* having the ALLOCATABLE attribute cannot be a dummy argument or a function result.

An array must not have both the ALLOCATABLE attribute and the POINTER attribute.

If an *entity* has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

An *entity* may not be given the same attribute more than once in a scoping unit.

If *char-length* is a non-constant expression, the length is declared at the entry of the procedure and is not affected by any redefinition of the variables in the specification expression during execution of the procedure.

Example

```
character :: c1                           ! length one
character(len=4) :: c3="Yow!"
character(len=*),parameter :: c6=" Howdy" ! length six
character(len=3),dimension(2) :: ca2=(/" Fo","lks"/)
character :: ca4*4(2) ! ca4 has len=4, dimension=2
```

CLOSE Statement

Description

The CLOSE statement terminates the connection between a specified input/output unit number and an external file.

Syntax

CLOSE (*close-spec-list*)

Where:

close-spec-list is a comma-separated list of *close-spec*s.

close-specs are:
*[*UNIT =*] external-file-unit,*
*[*IOSTAT=*iostat],*
*[*ERR=*label],*
*[*STATUS=*status]*

external-file-unit is the input/output unit number of an external file.

iostat is a scalar default INTEGER variable. It signals either success or failure after execution of the CLOSE statement.

label is the label of a branch target statement to which the program branches if there is an error in executing the CLOSE statement.

status is a CHARACTER expression that evaluates to either 'KEEP' or 'DELETE'.

Remarks

The CLOSE statement must specify an *external-file-unit.*

If UNIT=is omitted, *external-file-unit* must be the first specifier in *close-spec-list*.

If IOSTAT=*iostat* is present, *iostat* has a value of zero if the unit was successfully closed. If an error occurs when executing the CLOSE statement, *iostat* is assigned an error number which identifies which error occurred.

A specifier must not appear more than once in a CLOSE statement.

STATUS='KEEP' must not be specified for a file whose status is ’SCRATCH’. If ’KEEP’ is specified for a file that exists, the file continues to exist after a CLOSE statement. This is the default behavior.

If STATUS='DELETE' is specified, the file associated with the unit number will be deleted upon execution of the CLOSE statement.

Example 1

```
integer :: ios
close(8,iostat=ios,status='DELETE')
if(ios == 0) then
  write(*,*) " No error occurred."
else
  write(*,*) " IOSTAT= encourages structured programming."
end if
```

Example 2

```
close(unit=8,err=200)
write(*,*) " No error occurred."
goto 300
200 write(*,*) " An error occurred"
300 continue
```

CMPLX Function

Description

The CMPLX function uses REAL or INTEGER arguments to compose a result of type COMPLEX. It may also convert between different kinds of COMPLEX numbers, possibly resulting in a loss of precision.

Syntax

CMPLX (*x [, y] [, kind]*)

Required Arguments

x is an INTENT(IN) scalar or array of type REAL, INTEGER, or COMPLEX.

Optional Arguments

y is INTENT(IN) and of type REAL or INTEGER. If *x* is of type COMPLEX, *y* cannot be present.

kind determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *“KIND Function”*, or *“SELECTED_REAL_KIND Function”*.

Result

The result is of type COMPLEX.

If *x* is INTEGER or REAL, the value of the result is the complex number whose real part has the value of *x*, and whose imaginary part has the value of *y*. If *y* is absent, the imaginary part of the result is zero.

If *x* is COMPLEX, it is as if *x* and *y* were present with the values REAL(*x*), AIMAG(*x*)).

If *kind* is present, the result is of the kind specified by *kind*.

If *kind* is absent, the result is default kind.

Example

```
real :: x=1.,y=1.
integer :: ix=1,iy=1
complex(kind(1.d0)) :: z=(1.d0,1.d0)
write(*,*) cmplx(x)    ! y assumed to be zero
write(*,*) cmplx(x,y)
write(*,*) cmplx(ix,iy,kind(1.d0))
write(*,*) cmplx(ix,y)
write(*,*) z, cmplx(z) ! precision is lost
```

COMMON Statement

Description

The COMMON statement provides a global data facility. It specifies contiguous blocks of physical storage, called common blocks, that are available to any program unit that references the common block.

Syntax

COMMON *[/ [common-name] /] common-object-list [[,] / [common-name] / common-object-list] ...*

Where:

common-name is the name of the common block being declared.

common-object-list is a comma-separated list of data objects that are to be included in the common block.

Remarks

If *common-name* is present, all data objects in the corresponding *common-object-list* are specified to be in the named common block *common-name*.

If *common-name* is omitted, all data objects in the following *common-object-list* are specified to be in blank common.

For each common block, a contiguous storage sequence is formed for all data objects, in the order they appear in *common-object-list*s in the program unit.

A given data object can appear only once in all *common-object-list*s in a program unit.

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 to become undefined unless the common block name has been declared in a SAVE statement.
2. Named common blocks of the same name must be the same size in all scoping units of a program in which they appear, but blank commons can be of different sizes.
3. A data object in named common can be initialized in a BLOCK DATA program unit, but data objects in a blank common must not be initially defined.

A common block name or blank common can appear multiple times in one or more COMMON statements in a program unit. In such case, the *common-object-list* is treated as a continuation of the *common-object-list* for that common block.

A data object in a *common-object-list* must not be a dummy argument, an allocatable array, an automatic object, a function name, an entry name, or a result name, and it must have a name made available by use association.

Each bound in an array-valued data object in a *common-object-list* must be an initialization expression.

Any data object must only become associated with an object having the same attributes, type, kind, length, or rank.

If a data object in a *common-object-list* has an explicit shape, it cannot have the pointer attribute.

If a data object in a *common-object-list* is of a derived type, the derived type must have the sequence attribute.

Derived type data objects in which all components are of default numeric or LOGICAL types can become associated with data objects of default numeric or LOGICAL types.

Derived type data objects in which all components are of default CHARACTER type can become associated with data objects of type CHARACTER.

An EQUIVALENCE statement must not cause the storage sequences of two different common blocks to become associated.

An EQUIVALENCE statement must not cause storage units to be added before the first storage unit of the common block.

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.

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 appended to /first/
```

COMPLEX Statement

Description

The COMPLEX statement declares entities having the COMPLEX data type.

Syntax

COMPLEX *[kind-selector] [[, attribute-list] ::] entity [, entity] ...*

Where:

kind-selector is (*[*KIND=*] scalar-int-initialization-expr*)

scalar-int-initialization-expr is a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, the value of this descriptor should be the result of a *"KIND Function"* or a *"SELECTED_REAL_KIND Function"*.

attribute-list is a comma-separated list from the following attributes: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), EXTERNAL, INTENT(IN) or INTENT(OUT) or INTENT(IN OUT), PUBLIC or PRIVATE, INTRINSIC, OPTIONAL, POINTER, SAVE, TARGET, VOLATILE, DLL_EXPORT or DLL_IMPORT or ML_EXTERNAL.

entity is *entity-name [(array-spec)] [=initialization-expr]*
or *function-name [(array-spec)]*

array-spec is an array specification.

initialization-expr is an expression that can be evaluated at compile time.

entity-name is the name of an entity being declared.

function-name is the name of a function being declared.

Remarks

function-name must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

The =*initialization-expr* must appear if the statement contains a PARAMETER attribute.

If =*initialization-expr* appears, a double colon must appear before the list of *entities*. Each *entity* has the SAVE attribute, unless it is in a named common block.

The =*initialization-expr* must not appear if *entity-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 pointer, an external name, an intrinsic name, or an automatic object.

If an array or function with an array result is declared with the POINTER attribute, it must have a deferred shape.

If an array is declared with the ALLOCATABLE attribute, it must have a deferred shape.

If an array or function with an array result does not have the POINTER or the ALLOCATABLE attribute, it must be specified with an explicit shape.

If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attributes must not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attributes must not be specified.

The PARAMETER attribute cannot be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

The INTENT(IN), INTENT(OUT), INTENT(IN OUT), and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* may not have the PUBLIC attribute if its type has the PRIVATE attribute.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

An *entity* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

A subprogram must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

An *entity* having the ALLOCATABLE attribute cannot be a dummy argument or a function result.

An array must not have both the ALLOCATABLE attribute and the POINTER attribute.

If an *entity* has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

An *entity* may not be given the same attribute more than once in a scoping unit.

Example

```
complex :: a, b
complex,parameter :: c=(2.0,3.14159)! c must be initialized
complex :: d(2)=(/(1.,0.),(0.,0.)/) ! array initialization
complex,pointer :: e(:,:,:)         ! deferred shape rank 3
```

Computed GOTO Statement (obsolescent)

Description

The computed GOTO statement causes transfer of control to one of a list of labeled statements.

Syntax

GO TO (*labels*) *[,] scalar-int-expr*

Where:

labels is a comma-separated list of labels.

scalar-int-expr is a scalar INTEGER expression.

Remarks

Execution of a computed GOTO statement causes evaluation of *scalar-int-expr*. If this value is *i* such that $1 \le i \le n$, 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.

Each label in *labels* must be the label of a branch target statement in the current scoping unit.

The computed GOTO statement has been identified as a major contributor to a logic-snarled condition known as "spaghetti code", which makes a program difficult to read and debug. The computed GOTO statement is best replaced by the *“CASE Construct”* although the *“IF Construct”* could be used as well. Although the computed GOTO statement is obsolescent and should never be used when writing new code, it is fully supported.

Example

```
   integer :: i=1
40 write(*,*) " computed goto construct"
   goto (20,30,40) i
   write(*,*) " transfer here if no match"
   goto 10
30 write(*,*) " if i=2 control transfers here"
20 write(*,*) " if i=1 control transfers here"
10 write(*,*) " equivalent case construct"
   select case (i)
   case(1)
     write(*,*) " if i=1 control transfers here"
   case(2)
     write(*,*) " if i=2 control transfers here"
   case(3)
     write(*,*) " if i=3 control transfers here"
   case default
     write(*,*) " transfer here if no match"
   end select
```

CONJG Function

Description

The CONJG function returns the conjugate of a complex number.

Syntax

CONJG (*z*)

Arguments

z is an INTENT(IN) scalar or array of type COMPLEX.

Result

The result is of type COMPLEX and has the same kind as *z*. Its value is *z* with the imaginary part negated.

Example

```
complex :: x=(.1,-.2),y(2)=(/(1.,0.),(0.,1.)/)
write(*,*) conjg(x) ! writes (.1, .2)
write(*,*) conjg(y) ! writes (1.,0.) (0.,-1.)
```

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

When used in a MODULE, the CONTAINS statement separates a global data area from any module procedures. Any variables, type definitions, interfaces or initialization expressions that appear above the CONTAINS statement are globally available within the module, and are also available to any program unit that uses the module, provided that the entity in question has the PUBLIC attribute.

When appearing in a main program, subprogram or module procedure, the CONTAINS statement separates the main body of code from any internal procedures. Any variables, type definitions, interfaces or initialization expressions that appear above the CONTAINS statement are available to all internal procedures that appear below the CONTAINS statement.

Any variables, type definitions, interfaces, or initialization expressions that appear below a CONTAINS statement are local in scope.

The CONTAINS statement is not executable.

Internal procedures cannot contain other internal procedures.

Example

```
module mod1
  real :: a=1.   ! a is globally available
contains         ! separates global from procedures
  subroutine sub1()
    real :: b=3. ! b is only available inside sub1()
    write(*,*) a ! global a is available inside sub1()
    call internal()
  contains       ! separates sub1 body from internal proc
    subroutine internal() ! internal() is local to sub1
      real :: a=2. ! this a is local to internal()
      write(*,*) b ! b is available by host association
      write(*,*) a ! global a is not available because
                   ! it is overridden by local a
    end subroutine internal
  end subroutine sub1
end module mod1
```

```
program prog1
  use mod1
  call internal()
contains
  subroutine internal()
    write(*,*) a ! global a is available by host association
    call sub1()
  end subroutine internal
end program
```

CONTINUE Statement

Description

The CONTINUE statement is traditionally used in conjunction with a statement label, as the target of a branch statement or a do loop terminus. Execution of a CONTINUE statement has no effect.

Syntax

CONTINUE

Remarks

If a labeled CONTINUE statement marks the terminus of a do loop, it must not be the target of a branch statement and it cannot be used as the terminus of any other do loop.

Example

```
   integer :: i
20 continue
   do 10 i=1,10
10 continue
   goto 20
```

COS Function

Description

The COS function returns the trigonometric cosine of a REAL or COMPLEX argument.

Syntax

COS (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL or COMPLEX and must be expressed in radians.

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*.

Example

```
real :: x=.5,y(2)=(/1.,1./)
complex :: z=(1.,1.)
write(*,*) cos(x) ! writes .8775826
write(*,*) cos(y) ! writes .9950042 .9950042
write(*,*) cos(z) ! writes (.8337300. -.9888977)
```

COSH Function

Description

The COSH function returns the hyperbolic cosine of a REAL argument.

Syntax

COSH (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is of the same type and kind as *x*. Its value is a REAL representation of the hyperbolic cosine of *x*.

Example

```
real :: x=.5,y(2)=(/1.,1./)
write(*,*) cosh(x) ! writes 1.127626
write(*,*) cosh(y) ! writes 1.543081 1.543081
```

COUNT Function

Description

The COUNT function counts the number of true elements in a logical mask either for an entire mask or along a given dimension of the mask.

Syntax

COUNT (*mask [, dim]*)

Required Arguments

mask is an INTENT(IN) array of type LOGICAL. It must not be scalar.

Optional Arguments

dim is an INTENT(IN) scalar of type INTEGER with a value within the range $1 \leq dim \leq n$, where *n* is the rank of *mask*. The corresponding actual argument must not be an optional dummy argument.

Result

The result is of type default INTEGER. Its value and rank are computed as follows:

1. The function will return a scalar logical value if *mask* has rank one, or if *dim* is absent. The result is the number of elements for which *mask* is true.
2. The function will return a logical array of rank *n*-1 if *dim* is present and *mask* has rank two or greater. The resulting array is of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ 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*.

Example

```
integer,dimension(2,3) :: a,b
a=reshape((/1,2,3,4,5,6/),shape(a))
write(*,'(2i3)') a ! writes 1  2
                   !       3  4
                   !       5  6
b=reshape((/1,2,3,5,6,4/), (/2,3/))
write(*,'(2i3)') b ! writes 1  2
                   !       3  4
                   !       5  6
write(*,*) count(a==b)       ! writes 3
write(*,*) count(a==b,dim=1) ! writes 2 1 0
write(*,*) count(a==b,dim=2) ! writes 2 1
```

CPU_TIME Subroutine

Description

The CPU_TIME subroutine returns the amount of processor time used by a program, expressed as a REAL number.

Syntax

CPU_TIME (*time*)

Required Arguments

time is an INTENT(OUT) scalar REAL variable. It is assigned the processor time in seconds.

Remarks

CPU_TIME only reflects the actual CPU usage when the application is executed in an environment that makes this information available. Windows NT, 2000, XP and Linux systems support this facility. If the operating system does not track CPU usage, CPU_TIME returns the elapsed time between calls.

Example

```
integer :: i
real :: start_time,end_time, x(1000000)
call cpu_time(start_time)
do i=1,1000000
  x=cosh(real(i))
end do
call cpu_time(end_time)
write(*,*) end_time-start_time ! writes elapsed time
```

CSHIFT Function

Description

The CSHIFT function performs a 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.

Syntax

CSHIFT (*array*, *shift [*, *dim]*)

Required Arguments

array is an INTENT(IN) array of any type. It must not be scalar.

shift is an INTENT(IN) INTEGER and must be scalar if *array* is of rank one; otherwise it can either be scalar or of rank *n*-1 and shape $(d_1, d_2, \dots, d_{dim-1}, d_{dim+1}, \dots, d_n)$, where $(d_1, d_2, \dots, d_n)$ is the shape of *array*.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER with a value in the range $1 \le dim \le n$, 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* + *n* - 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 an array.

Example

```
integer :: a(3), b(3,3)
a = (/1,2,3/)
b = reshape ((/1,2,3,4,5,6,7,8,9/), (/3,3/))
write(*,10) a                     ! writes 1 2 3
write(*,10) cshift(a, 1)          ! writes 2 3 1
write(*,20) b                     ! writes 1 2 3
                                  !        4 5 6
                                  !        7 8 9
write(*,20) cshift(b,-1)          ! writes 3 1 2
                                  !        6 4 5
                                  !        9 7 8
write(*,20) cshift(b,(/1,-1,0/))! writes 2 3 1
                                  !        6 4 5
                                  !        7 8 9
write(*,20) cshift(b,1,dim=2)     ! writes 4 5 6
                                  !        7 8 9
                                  !        1 2 3
10 format(3i3)
20 format(3(/,3i3))
```

CYCLE Statement

Description

The CYCLE statement skips to the next 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. If *do-construct-name* is omitted, it is as if *do-construct-name* were the name of the innermost DO construct in which the CYCLE statement appears.

Remarks

The CYCLE statement may only appear within a DO construct.

Example

```
       integer :: i, j
outer: do i=1, 10
         if(i < 3) cycle          ! cycles outer
inner:   do j=1, 10
           if (i < j) cycle       ! cycles inner
           if (i > j) cycle outer ! cycles to outer
         end do inner
       end do outer
```

DATA Statement

Description

The DATA statement provides initial values for data objects.

Syntax

DATA *data-stmt-set [[,] data-stmt-set] ...*

Where:

data-stmt-set is *object-list* / *value-list* /

object-list is a comma-separated list of variable names or *implied-do*s.

value-list is a comma-separated list of *[repeat *] data-constant*

repeat is a scalar INTEGER constant.

data-constant is a scalar constant (either literal or named)
or a structure constructor.

implied-do is (*implied-do-object-list* , *implied-do-var=expr*, *expr[*, *expr]*)

implied-do-object-list is a comma-separated list of array elements, scalar structure components, or *implied-do*s.

implied-do-var is a scalar INTEGER variable.

expr is a scalar INTEGER expression.

Remarks

Each object in *object-list* must have a corresponding value in *value-list*.

Each value in *value-list* must be a constant that is either previously defined or made accessible by host or use association. Each constant should be of the same kind as the corresponding object being initialized.

A variable, or part of a variable, must not be initialized more than once in an executable program.

If the type of a variable that is being initialized is not declared prior to its appearance in a DATA statement, it is of default type. Any subsequent declaration of the type of the variable must be of default kind.

A whole array that appears in an *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.

An *implied-do* is expanded to form a sequence of array elements and structure components, under the control of the *implied-do-var*, as in the DO construct.

repeat indicates the number of times the following constant is to be included in the sequence; omission of *repeat* defaults to a repeat factor of 1.

A variable that is initialized in a DATA statement cannot also be any of the following: a dummy argument; accessible by host or use association; in a blank common block; be a function name or function result name; an automatic object; a pointer; or an allocatable array.

Variables that are initialized using the DATA statement in a block data program unit may appear in a named common block. Variables that are initialized using the DATA statement in program units other than block data cannot appear in a named common block.

If an object in an *object-list* is of type INTEGER, its corresponding value may be a binary, octal, or hexadecimal constant.

Example

```
integer, parameter :: arrsize=100000,init=0
real,parameter :: rinit=0.
real :: r1,r2,r3,array1(2,2),array2(arrsize)
real(kind(1.d0)) :: r4,r5
complex :: q
integer :: l,b,o,z,array3(10)
data r1,r2,r3 /1.,2.,3./, array1 /1.,2.,3.,4./
data r4 /1.23456789012345d0/ ! correct initialization
data r5 /1.23456789012345/   ! loses precision
data array2 /arrsize*rinit/,q /(0.,0.)/
data (array3(l),l=1,10) /10*init/
data b /B'01101000100010111110100001111010'/
data o /O'15042764172'/
data z /Z'688be87a'/
write(*,*) r4,r5
```

DATE_AND_TIME Subroutine

Description

The DATE_AND_TIME subroutine retrieves information concerning a computer's calendar date, time of day and time zone at the time the subroutine is invoked.

Syntax

DATE_AND_TIME (*[date [,]] [time[,]] [zone[,]] [values[,]]*)

Optional Arguments

date is an INTENT(OUT) scalar of type CHARACTER, with a minimum length of eight. Its leftmost eight characters are set to a value of the form *yyyymmdd*, where *yyyy* is the year, *mm* the month, and *dd* the day. If there is no date available, *date* is blank.

time is an INTENT(OUT) scalar of type CHARACTER, with a minimum length of ten. Its leftmost ten characters are set to a value of the form *hhmmss.sss*, where *hh* is the hour, *mm* the minutes, and *ss.sss* is seconds and milliseconds. If there is no clock available, *time* is blank.

zone is an INTENT(OUT) scalar of type CHARACTER, with a minimum length of five. 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 minutes. If there is no clock or time zone information available, *zone* is blank.

values an INTENT(OUT) rank one array of type default INTEGER with a minimum size of eight. If any date or time value is unavailable, its corresponding element in *values* will be set to -huge(0). Otherwise, the first eight elements of *values* are as follows:

values (1) the year (for example, 2002)

values (2) the month of the year

values (3) the day of the month

values (4) the time difference with respect to Coordinated Universal Time (UTC) in minutes

values (5) the hour of the day, in the range of 0 to 23

values (6) the minutes of the hour, in the range of 0 to 59

values (7) the seconds of the minute, in the range 0 to 59

values (8) the milliseconds of the second, in the range 0 to 999

Remarks

If keyword arguments are not used, the date value will always be returned in the first argument of the calling list, time in the second argument, zone in the third argument, and values in the fourth argument.

Example

```
character(len=10) :: time,date,zone
integer :: dt(8)
call date_and_time(time=time) ! keyword arguments
call date_and_time(date=date) !
call date_and_time(zone=zone) !
write(*,*) time,date,zone
call date_and_time(date,time,zone) ! positional arguments
write(*,*) time,date,zone
call date_and_time(values=dt)
write(*,*) dt
call date_and_time(date) ! if no keywords are used, always
call date_and_time(time) ! returns the date
call date_and_time(zone) !
write(*,*) time,date,zone
```

DBLE Function

Description

The DBLE function returns a double-precision REAL value given a numeric argument.

Syntax

DBLE (*a*)

Arguments

a is an INTENT(IN) scalar or array of type INTEGER, REAL or COMPLEX.

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*.

Example

```
integer :: i=1
real :: r=1.
complex :: q=(1.,1.)
write(*,*) i,dble(i)
write(*,*) r,dble(r)
write(*,*) q,dble(q)
```

DEALLOCATE Statement

Description

The DEALLOCATE statement deallocates allocatable arrays and pointer targets and disassociates pointers.

Syntax

DEALLOCATE (*object-list [*, STAT=*stat-variable]*)

Where:

object-list is a comma-separated list of pointers or allocatable arrays.

stat-variable is a scalar INTEGER variable that returns a status value.

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 a DEALLOCATE statement that does not contain the STAT= specifier, the executable program is terminated.

Deallocating an allocatable array that is not currently allocated or a pointer that is disassociated or whose target was not allocated causes an error condition in the DEALLOCATE statement.

If a pointer is currently associated with an allocatable array, the pointer must not be deallocated.

Deallocating an allocatable array or pointer causes the status of any pointer associated with it to become undefined.

Example

```
integer,pointer,dimension(:,:) :: ip => null()
integer,allocatable,dimension(:) :: jp
integer :: allostat
allocate (ip(10,20),jp(10))
deallocate(ip)
deallocate(jp,stat=allostat)
write(*,*) allostat
```

DIGITS Function

Description

The DIGITS function returns the number of significant binary digits in a real or integer data object.

Syntax

DIGITS (*x*)

Arguments

x an INTENT(IN) scalar or array of type INTEGER or REAL.

Result

The result is of type default INTEGER. Its value is the number of binary digits composing the significant value of *x*.

Example

```
real :: r
real(kind(1.d0)) :: d
integer :: i
write(*,*) digits(i),digits(r),digits(d)
```

DIM Function

Description

The DIM function returns the difference between two numbers if the difference is positive; zero otherwise.

Syntax

DIM (*x*, *y*)

Arguments

x is an INTENT(IN) scalar or array of type INTEGER or REAL.

y is INTENT(IN) and 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.

Example

```
integer :: i=1,j=2
real :: x=1.,y=.5
write(*,*) dim(x,y) ! writes 0.5
write(*,*) dim(i,j) ! writes 0.0
```

DIMENSION Statement

Description

The DIMENSION statement specifies the shape or rank 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-specs*
or *assumed-shape-specs*
or *deferred-shape-specs*
or *assumed-size-spec*

explicit-shape-specs is a comma-separated list of *[lower-bound* :*] upper-bound* that specifies the shape and bounds of an explicit-shape array.

assumed-shape-specs is a comma-separated list of *[lower-bound]* : that, with the dimensions of the corresponding actual argument, specifies the shape and bounds of an assumed-shape array.

deferred-shape-specs is a comma-separated list of colons that specifies the rank of a deferred-shape array.

assumed-size-spec is *[explicit-shape-specs,] [lower-bound* :*]* *

assumed-size-spec specifies the shape of a dummy argument array whose size is assumed from the corresponding actual argument array.

lower-bound is a scalar INTEGER expression that can be evaluated on entry to the program unit that specifies the lower bound of a given dimension of the array.

upper-bound is a scalar INTEGER expression that can be evaluated on entry to the program unit that specifies the upper bound of a given dimension of the array.

Remarks

If the object being dimensioned also has the ALLOCATABLE or POINTER attribute, *array-spec* must be specified as a *deferred-shape*.

Example 1

```
program prog1
  dimension :: a(3,2,1)       ! dimension statement
  real, dimension(3,2,1) :: b ! dimension attribute
  dimension c(-3:3)           ! bounds specified
  real d
  allocatable d
  dimension d(:,:,:)          ! deferred shape with rank 3
```

Example 2

```
subroutine sub1(x,y,z)
  dimension :: x(:,:,:) ! assumed shape with rank 3
  dimension y(-3:)      ! lower bound specified
  dimension z(*)        ! assumed size array
```

DLL_EXPORT Statement

Description

The DLL_EXPORT statement makes a procedure that resides in a dynamic-link library externally available.

Syntax

DLL_EXPORT *[::] dll-export-names*

Where:

dll-export-names is a list of procedures defined in the current scoping unit.

Remarks

The procedures in *dll-export-names* must not be module procedures.

The procedures names listed in a DLL_EXPORT statement are "decorated" to match one of several calling conventions by using the "-ml xxxx" switch at compile time.

Example

```
function half(x)
  integer :: half,x
  dll_export half             ! dll_export statement
  half=x/2
end function half
function twice(x)
  integer,dll_export :: twice ! dll_export attribute
  integer :: x
  twice=x*2
end function twice
```

DLL_IMPORT Statement

Description

The DLL_IMPORT statement specifies which procedures are to be imported from a dynamic-link library.

Syntax

DLL_IMPORT *[::] dll-import-names*

Where:

dll-import-names is a comma-separated list of procedure names.

The procedures names listed in a DLL_IMPORT statement are "decorated" to match one of several calling conventions by using the "-ml xxxx" switch at compile time.

Example

```
program main
  integer :: half,i
  dll_import half              ! dll_import statement
  integer,dll_import :: twice ! dll_import attribute
  i=half(i)
end program main
```

DO Construct

Description

The DO construct specifies the repeated execution (loop) of a block of code.

A DO statement begins a DO construct.

An END DO statement ends the innermost nested DO construct.

Syntax

[construct-name :*]* DO *[label] [loop-control]*

block

do-termination

Where:

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=expr*, *expr [*, *expr]*
or *[,]* WHILE (*while-expr*)
do-variable is a scalar variable of type INTEGER.

expr is a scalar expression of type INTEGER. The first *expr* is the initial value of *do-variable*; the second *expr* is the final value of *do-variable*; the third *expr* is the increment value for *do-variable*.
while-expr is a scalar LOGICAL expression.
block is a sequence of zero or more statements or executable constructs.
do-termination is END DO *[construct-name]*
or *label action-stmt*
action-stmt is a statement other than GOTO, RETURN, STOP, EXIT, CYCLE, assigned GOTO, arithmetic IF, or END.

Remarks

If a *do-variable* is present, the expressions in are evaluated, and *do-variable* is assigned an initial value and an iteration count. An iteration count of zero is possible. Note that because the iteration count is established before execution of the loop, changing the *do-variable* within the range of the loop has no effect on the number of iterations.

If *loop-control* is WHILE (*while-expr*), *while-expr* is evaluated and if false, the loop terminates.

If there is no *loop-control* it is as if the iteration count were effectively infinite.

Use of default or double-precision REAL for the *do-variable* has been removed from the Fortran 95 language.

The *"CYCLE Statement"* skips to the next iteration of a DO loop.

The *"EXIT Statement"* exits a DO loop altogether.

If the DO statement specifies a label, the corresponding *do-termination* statement must be identified with the same label.

If a construct name is specified in the DO statement, the same construct name must be specified in a corresponding END DO statement.

If the DO statement is not identified by a *construct-name*, the *do-termination* statement must not specify a *construct-name*.

Ending a DO construct with a labeled action statement is obsolescent, the use of END DO is preferred.

Example 1

```
integer :: i
real :: a=20.,b=10.
do i=1,10
                 ! code block goes here
end do
do               ! infinite do loop
                 ! better have some way to leave
   exit
```

```
end do
do while (a > b) ! does while condition is true
  a=a-1.
  write(*,*) a > b
end do
do i=10,1,-1     ! backward iteration of index i
                 ! code block goes here
end do
```

Example 2

```
integer :: i, j
outer_loop: do i=1,5
   inner_loop: do j=1,5
                  if(i>j) then
                     write(*,*) ' cycling inner'
                     cycle inner_loop
                  else if (i<j) then
                     write(*,*) ' cycling outer'
                     cycle outer_loop
                  else
                     write(*,*) i,j
                  end if
               end do inner_loop
            end do outer_loop
```

Example 3

```
    integer :: i
    do 10, i=1,10
10  end do          ! label number required
    do i=1,10
    end do
lp: do while(i>10)
        i=i-1
    end do lp       ! construct name required
```

Example 4

```
integer :: i
real :: a=20., b=10.
do i=1,5
   write(*,*) 'simple indexed do'
end do
do i=1,5,2
   write(*,*) 'indexed do with stride'
end do
do i=5,1
   write(*,*) 'zero trip loop'
end do
```

```
do while (a > b) ! does while condition is true
  a=a-1.
  write(*,*) a > b
end do
```

DOT_PRODUCT Function

Description

The DOT_PRODUCT function returns the dot product of two vectors of type INTEGER, REAL OR COMPLEX.

Syntax

DOT_PRODUCT (*vector_a*, *vector_b*)

Arguments

vector_a is an INTENT(IN) rank one array of type INTEGER, REAL, COMPLEX, or LOGICAL.

vector_a is INTENT(IN) and the same size as *vector_b*.

If *vector_a* is a numeric type, *vector_b* must also be a numeric type.

If *vector_a* is LOGICAL, *vector_b* must also be LOGICAL.

Result

If both *vector_a* and *vector_b* are REAL or INTEGER, the result is equal to
SUM (*vector_a* * *vector_b*) .

If either argument is of type COMPLEX, the result value is
SUM (CONJG (*vector_a*) * *vector_b*) .
If one of the arguments is not COMPLEX, it is treated as if it were complex with an imaginary part of zero.

If the arguments are of type LOGICAL, then the result value is
ANY (*vector_a* .AND. *vector_b*).

If the arguments are of different numeric types, the result type is taken from the argument with the higher type, where COMPLEX is higher than REAL, and REAL is higher than INTEGER.

The kind of the result is taken from the argument that offers the greatest range.

If the argument arrays size is zero, the result is zero for numeric types, and false for logical types.

Example

```
integer :: ivec(3)=(/1,2,3/), &
           jvec(3)=(/4,5,6/)
real :: rvec(3)=(/1.,2.,3./)
real(kind(1.d0)) :: svec(3)=(/4.d0,5.d0,6.d0/)
complex :: pvec(3)=(/(0.,1.),(0.,2.),(0.,3.)/), &
           qvec(3)=(/(0.,4.),(0.,5.),(0.,6.)/)
write(*,*) dot_product(ivec,jvec) ! integer result
write(*,*) dot_product(rvec,jvec) ! real result
write(*,*) dot_product(rvec,svec) ! D.P. result
write(*,*) dot_product(pvec,jvec) ! Complex result
write(*,*) dot_product(pvec,svec) ! D.P. complex result
write(*,*) dot_product(pvec,qvec) ! Complex result
```

DOUBLE PRECISION Statement

Description

The DOUBLE PRECISION statement declares entities of type double precision REAL.

Syntax

DOUBLE PRECISION *[[, attribute-list]* ::*] entity [, entity]* ...

Where:

attribute-list is a comma-separated list from the following attributes: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), EXTERNAL, INTENT(IN) or INTENT(OUT) or INTENT(IN OUT), PUBLIC or PRIVATE, INTRINSIC, OPTIONAL, POINTER, SAVE, TARGET, VOLATILE, DLL_EXPORT or DLL_IMPORT or ML_EXTERNAL.

entity is *entity-name [*(*array-spec*)*] [=initialization-expr]*
or *function-name [*(*array-spec*)*]*

array-spec is an array specification.

initialization-expr is an expression that can be evaluated at compile time.

entity-name is the name of an entity being declared.

function-name is the name of a function being declared. It must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

Remarks

function-name must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

The =*initialization-expr* must appear if the statement contains a PARAMETER attribute.

If =*initialization-expr* appears, a double colon must appear before the list of *entities*. Each *entity* has the SAVE attribute, unless it is in a named common block.

The =*initialization-expr* must not appear if *entity-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 pointer, an external name, an intrinsic name, or an automatic object.

If an array or function with an array result is declared with the POINTER attribute, it must have a deferred shape.

If an array is declared with the ALLOCATABLE attribute, it must have a deferred shape.

If an array or function with an array result does not have the POINTER or the ALLOCATABLE attribute, it must be specified with an explicit shape.

If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attributes must not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attributes must not be specified.

The PARAMETER attribute cannot be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

The INTENT(IN), INTENT(OUT), INTENT(IN OUT), and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* may not have the PUBLIC attribute if its type has the PRIVATE attribute.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

An *entity* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

A subprogram must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

An *entity* having the ALLOCATABLE attribute cannot be a dummy argument or a function result.

An array must not have both the ALLOCATABLE attribute and the POINTER attribute.

If an *entity* has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

An *entity* may not be given the same attribute more than once in a scoping unit.

Example

```
double precision a,b(10)
double precision,dimension(2,4) :: d
double precision :: e=2.0d0
```

DPROD Function

Description
The DPROD function returns a double precision REAL product, given two single precision REAL arguments.

Syntax
DPROD (*x*, *y*)

Arguments
x is an INTENT(IN) scalar or array of type default REAL.

y is INTENT(IN) and scalar if *x* is a scalar, or an array if *x* is an array. *y* is of type default REAL.

Result
The result is of type double-precision REAL. Its value is an approximation of the double-precision product of *x* and *y*.

Example
```
real :: x=1.25,y=1.25
write(*,*) x*y,dprod(x,y) ! writes 1.56250000000000
```

DVCHK Subroutine (Windows Only)

Description
The DVCHK subroutine masks and detects divide by zero exceptions.

Syntax
DVCHK (*lflag*)

Arguments
lflag must be a scalar of type LOGICAL.

lflag must be set to true on the first invocation.

On subsequent invocations *lflag* is assigned the value true if a divide-by-zero exception has occurred, and false otherwise.

Remarks
The initial invocation of the DVCHK subroutine masks the divide-by-zero interrupt on the floating-point unit.

DVCHK will not check or mask zero divided by zero. Use INVALOP to check for a zero divided by zero.

Example

```
logical :: lflag=.true.
call dvchk(lflag)  ! mask the divide-by-zero interrupt
write(*,*) lflag   ! writes F
write(*,*) 1./0.   ! writes Inf
call dvchk (lflag)
write(*,*) lflag   ! writes T
```

ELEMENTAL Procedure

Description

An ELEMENTAL procedure declaration implies that the procedure may be called using scalar or array arguments.

Syntax

ELEMENTAL SUBROUTINE *subroutine-name* (*[dummy-arg-names]*)

or

ELEMENTAL *[type-spec]* FUNCTION *function-name* (*[dummy-arg-names]*) *[*RESULT *(result-name)]*

Where:

subroutine-name is the name of the subroutine.

dummy-arg-names is a comma-separated list of dummy argument names.

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 TYPE (*type-name*)

kind-selector is (*[*KIND=*] kind*)

char-selector is (*[*LEN=*] length [*, KIND=*kind]*)
or (KIND=*kind [*, LEN=*length]*)
or * *char-length [,]*

kind is a scalar INTEGER expression that can be evaluated at compile time.

length is a scalar INTEGER expression
or *

char-length is a scalar INTEGER literal constant
or (*)

function-name is the name of the function.

result-name is the name of the result variable.

Remarks

Declaring a procedure to be ELEMENTAL also implies that the procedure is PURE. Elemental procedures are subject to all the restrictions of a *"PURE Procedure"*.

All dummy arguments and function results must be scalar, and cannot have the POINTER attribute, be a dummy procedure, or an alternate return.

Dummy arguments may not appear in a specification statement, except as an argument to one of the following functions: BIT_SIZE, KIND, LEN, DIGITS, EPSILON, HUGE, MAXEXPONENT, MINEXPONENT, PRECISION, RANGE, RADIX or TINY.

When calling an elemental procedure, all actual arguments must be conformable to each other.

Dummy arguments of elemental functions must have the INTENT(IN) attribute.

If any actual argument to an elemental subroutine is an array, all INTENT(OUT) and INTENT(IN OUT) arguments must also be an array and all arrays must be conformable.

Result

If all actual arguments to an elemental function are scalar, the result is scalar.

If any actual argument is an array, the result is an array and conformable with the array argument. The resulting array contains the value of the scalar operation performed on each element of the array.

Example

```
module mod1 ! gives us an implicit interface
contains
 elemental function elefun1(a,b)
    integer :: elefun1
    integer,intent(in) :: a,b
    elefun1=a-b
  end function elefun1
  elemental subroutine elesub1(a,b,c)
    integer,intent(out) :: c
    integer,intent(in) :: a,b
    c=a-b
  end subroutine elesub1
end module

program prog1
  use mod1
  integer :: i=0,j=-1,k,ia(3)=(/1,2,3/),ib(3)=(/4,5,6/),ic(3)
  write(*,*) elefun1(i,j)   ! writes 1.0
  write(*,*) elefun1(i,ia)  ! writes -1.0 -2.0 -3.0
  write(*,*) elefun1(ia,i)  ! writes 1.0 2.0 3.0
  write(*,*) elefun1(ia,ib) ! writes -3.0 -3.0 -3.0
  call elesub1(i,j,k)
  write(*,*) k          ! writes 1
  call elesub1(ia,j,ic)
  write(*,*) ic         ! writes 2 3 4
  call elesub1(i,ib,ic)
  write(*,*) ic         ! writes -4 -5 -6
  call elesub1(ia,ib,ic)
  write(*,*) ic         ! writes -3 -3 -3
end program
```

END Statement

Description

The END statement signals the end of a program unit.

Syntax

END *[class [name]]*

Where:

class is either PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA.

name is the name of the program unit.

Remarks

Executing an END statement within a function or subroutine is equivalent to executing a RETURN statement.

Executing an END statement within a main program unit terminates execution of the program.

Each program unit, module subprogram, or internal subprogram must have exactly one END statement.

If the program unit is a module procedure or an internal subprogram, *class* is required.

name can be used only if a name was given to the program unit in a PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement.

If *name* is present, it must be identical to the name specified in the PROGRAM, FUNCTION, SUBROUTINE, MODULE or BLOCK DATA statement.

The END PROGRAM, END FUNCTION, and END SUBROUTINE statements are executable and can be branch target statements.

The END MODULE, and END BLOCK DATA statements are non-executable.

Example

```
module mod1
contains
  subroutine modsub()
  end subroutine ! program class is required on module proc
end module       ! "module" is optional
program endtest
  use mod1
  call sub1()
  call modsub()
  call intsub()
  contains
  subroutine intsub()
  end subroutine    ! "subroutine" for internal procedure
end program endtest ! "program" and "endtest" are optional
subroutine sub1()
end
```

ENDFILE Statement

Description

The ENDFILE statement writes an endfile record to the specified unit as the next record of a file. The file pointer is then positioned after the end of the file.

Syntax

ENDFILE *unit-number*

or

ENDFILE (*position-spec-list*)

Where:

unit-number is a scalar INTEGER expression corresponding to the input/output unit number connected to an external file.

position-spec-list is *[[*UNIT =*] unit-number][*, ERR=*label][*, IOSTAT=*stat]* where UNIT=, ERR=, and IOSTAT= can be in any order but if UNIT= is omitted, then *unit-number* must be first.

label is a statement label that is branched to if an error condition occurs during execution of the statement.

stat is of type INTEGER and returns a status indicator.

Remarks

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be executed to reposition the file before any further data transfer occurs.

If IOSTAT is present, execution continues after an error or end condition is encountered, and *stat* is set to a non-zero number. The value of *stat* is set to zero if the write operation was successful. *stat* is assigned a positive value if an error condition occurs, and a negative value if an end-of-file, or end-of-record condition occurs.

If *unit-number* is connected to the console, the ENDFILE statement has no effect.

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.

Note that ENDFILE may only be used on sequential access files.

Example

```
  integer :: ios
  endfile 10
  endfile(unit=11,iostat=ios)
  endfile(12, err=20)
20 continue
```

ENTRY Statement

Description

The ENTRY statement permits a program unit to define multiple procedures, each with a different entry point.

Syntax

ENTRY *entry-name [*(*[dummy-arg-list]*) *[*RESULT (*result-name*)*]]*

Where:

entry-name is the name of the entry.

dummy-arg-list is a comma-separated list of dummy arguments or * alternate return indicators.

result-name is a variable containing a function result.

Remarks

An ENTRY statement can appear only in a subroutine or function.

If the ENTRY statement is in a function, an additional function is defined by that subprogram named *entry-name*. If *result-name* is present, the result variable is named *result-name*. If *result-name* is absent, the result variable is named *entry-name*. The characteristics of the function result are specified by the result variable.

If the ENTRY statement is contained in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is *entry-name*.

The dummy arguments of an ENTRY subprogram are solely defined by the ENTRY statement's argument list.

A dummy argument may not appear in an executable statement before it is introduced in an ENTRY statement.

Any dummy argument not introduced by an ENTRY statement is considered undefined and may not be referenced within the scope of that ENTRY subprogram.

RESULT can be present only if the ENTRY statement is in a function subprogram.

If RESULT is specified, *entry-name* must not appear in any specification statement in the scoping unit of the function program.

If RESULT is specified, *result-name* cannot be the same as *entry-name*.

entry-name may not be a dummy argument, or appear in an EXTERNAL or INTRINSIC statement.

An ENTRY statement must not appear within an executable construct such as DO, IF CASE, etc.

A dummy argument can be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

If the subprogram unit containing the ENTRY statement is declared as RECURSIVE, PURE or ELEMENTAL, the subprogram defined by the ENTRY statement also has those attributes.

Example

```
program main
  call sub1()
  call sub1entry()
end program main
subroutine sub1()
  write(*,*) 'subroutine call executes this part'
entry sub1entry()
  write(*,*) 'both calls execute this part'
```

```
end subroutine sub1
```

EOSHIFT Function

Description

The EOSHIFT function performs an end-off shift of all rank one sections in an array. Elements are shifted out at one end and copies of a boundary value are shifted in at the other. Different sections can be shifted by different amounts and in different directions by using an array-valued shift.

Syntax

EOSHIFT (*array, shift [, boundary] [, dim]*)

Required Arguments

array is INTENT(IN) and can be of any type. It must not be scalar.

shift is an INTENT(IN) scalar or array of type INTEGER. If *array* is rank one, *shift* must be scalar; otherwise *shift* may either be scalar or of rank *n*-1 and shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$, where $(d_1, d_2, \ldots, d_n)$ is the shape of *array*.

Optional Arguments

boundary is INTENT(IN) and of same type and kind as *array*. It must be scalar if *array* is of rank one. Otherwise it may be scalar or of rank *n*-1 and shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$.
If *array* is of type CHARACTER, *boundary* must have the same length as *array*.

dim is an INTENT(IN) scalar INTEGER with a value in the range $1 \le dim \le n$, 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 the same type, kind and shape as *array*. The boundary value is assigned to any element of the array which does not have a source value.

If *array* is of rank one, the result is the value of *array* shifted by *shift* elements

If *shift* is scalar, and *array* is of rank two or greater, each element along dimension *dim* is shifted by *shift* elements.

If shift is an array, each element along dimension *dim* is shifted by the amount specified in the corresponding *shift* vector.

If *boundary* is absent, the default pad values are zero for numeric types, blanks for CHARACTER, and false for LOGICAL.

Example

```
integer :: a(3), b(3,3)
a = (/1,2,3/)
b = reshape ((/1,2,3,4,5,6,7,8,9/), (/3,3/))
write(*,10) a                         ! writes 1  2  3
write(*,10) eoshift(a, 1)             ! writes 2  3  0
write(*,10) eoshift(a,1, -1)          ! writes 2  3 -1
write(*,20) b                         ! writes 1  2  3
                                      !        4  5  6
                                      !        7  8  9
write(*,20) eoshift(b,-1)             ! writes 0  1  2
                                      !        0  4  5
                                      !        0  7  8
write(*,20) eoshift(b,-1,(/1,-1,0/)) ! writes 1  1  2
                                      !       -1  4  5
                                      !        0  7  8
write(*,20) eoshift(b,(/1,-1,0/))    ! writes 2  3  0
                                      !        0  4  5
                                      !        7  8  9
write(*,20) eoshift(b,1,dim=2)        ! writes 4  5  6
                                      !        7  8  9
                                      !        0  0  0
10 format(3i3)
20 format(3(/,3i3))
```

EPSILON Function

Description

The EPSILON function returns a positive real value that is almost negligible compared to unity. It is the smallest value of *x* such that 1.+*x* is not equal to 1.

Syntax

EPSILON (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

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*.

Example

```
real(kind(1.d0)) :: d
real(kind(1.e0)) :: r
    ! adding epsilon only changes the rightmost bit
write(*,*) 1.d0+epsilon(d)
    ! writes 1.000000000000000
write(*,'(2z18.16)') 1.d0,1.d0+epsilon(d)
    ! show bits
    ! writes 3FF0000000000000  3FF0000000000001
write(*,*) 1.e0+epsilon(r)
    ! writes 1.00000012
write(*,'(2z10.8)') 1.e0,1.e0+epsilon(r)
    ! show bits
    ! writes 3F800000  3F800001
```

EQUIVALENCE Statement

Description

The EQUIVALENCE statement specifies two or more aliases that share the same storage.

Syntax

EQUIVALENCE *equivalence-sets*

Where:

equivalence-sets is a comma-separated list of (*equivalence-objects*)

equivalence-objects is a comma-separated list of variables, array elements, or substrings.

Remarks

An *equivalence-object* must not be: made available by use association; a dummy argument; a pointer; a target; an allocatable array; a subobject of a non-sequence derived type; a subobject 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 named constant; a structure component; or a subobject of any of these objects.

If the equivalenced objects have different types or kinds, the EQUIVALENCE statement does not perform any 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 scalar properties.

If an *equivalence-object* is a derived type that is not a numeric sequence or CHARACTER sequence type, all of the objects in the equivalence set must be of that 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 *equivalence-set* must be of the same type with the same kind value.

A data object of type default CHARACTER can be equivalenced only with other objects of type default CHARACTER. The lengths of the equivalenced objects are not required to be the same.

An EQUIVALENCE statement can not specify that the same storage unit is to occur more than once in a storage sequence.

When one of the equivalence objects is initialized, all associated objects are considered to be initialized. Each equivalence set may have only one initialization value.

If an *equivalence-object* has the VOLATILE attribute, all *equivalence-objects* in the *equivalence-set* are volatile.

Example

```
real :: a=1.
real(kind(1.d0)) :: d
integer :: i
logical :: l
equivalence (a,d,i,l)
write(*,*) a ! writes 1.0000000
write(*,*) d ! writes 5.263544247120890E-315
write(*,*) i ! writes 1065353216
write(*,*) l ! writes T
```

ERROR Subroutine

Description

The ERROR subroutine prints an error message with traceback to the console and continues processing.

Syntax

ERROR (*message*)

Arguments

message is an INTENT(IN) argument of type CHARACTER. It contains the message to be printed.

Remarks

If the program is compiled with the -ntrace (Windows) or --ntrace (Linux) option, a traceback will not be printed.

Example

```
character(len=8) :: errmsg=' Error: '
call error(errmsg) ! writes Error:
                   ! followed by a traceback message
```

EXIT Statement

Description

The EXIT statement causes execution of a specified DO loop to be terminated. Execution continues at the first executable statement after the loop terminus.

Syntax

EXIT *[do-construct-name]*

Where:

do-construct-name is the name of a DO construct that contains the EXIT statement. If *do-construct-name* is omitted, the EXIT statement applies to the innermost DO construct in which the EXIT statement appears.

Example

```
       integer :: i, j
outer: do i=1, 10
inner:   do j=1, 10
           if (i < j) then
             exit               ! exits inner
           else if (i > j) then
             cycle
           else
             write(*,*) i,j
             exit outer
           end if
         end do inner
       end do outer
```

EXIT Subroutine

Description

The EXIT subroutine causes program execution to terminate with an exit code.

Syntax

EXIT (*ilevel*)

Arguments

ilevel must be a scalar of type INTEGER. It sets the value of the program's exit code.

Example

```
call exit(3)  ! exit -- system error level 3
```

EXP Function

Description

The EXP function returns a REAL or COMPLEX value that is an approximation of the exponential function.

Syntax

EXP (*x*)

Arguments

x is an INTENT(IN) scalar or array 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 e^x.

If *x* is COMPLEX, its imaginary part is treated as a value in radians.

Example

```
real :: r=1.
complex :: c=(0.,-3.141592654)
write(*,*) exp(r) ! writes an approximation of e
write(*,*) exp(c) ! writes a complex approximation of -1.
```

EXPONENT Function

Description

The EXPONENT function returns the exponential part of the model representation of a number.

Syntax

EXPONENT (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is of type default INTEGER. Its value is the power of two of the exponential part of *x*.

Example

```
real :: x=4.3
write(*,*) x,exponent(x)
           ! writes 4.300000 3
write(*,*) scale(fraction(x),exponent(x))
           ! writes 4.300000
```

EXTERNAL Statement

Description

The EXTERNAL statement declares external procedures. Specifying a procedure name as EXTERNAL permits the procedure 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.

A name can appear only once in all of the EXTERNAL statements in a scoping unit.

If the external procedure is a block data subprogram, the inclusion of the block data in the program is required.

Example

```
program main
  external sub1             ! external statement
  integer,external :: fun1 ! external attribute
  call bill(sub1,fun1)
end program
subroutine bill(proc,fun)
  integer :: fun,i
  i=fun()
  call proc(i)
end subroutine
subroutine sub1(i)
  integer :: i
  write(*,*) i
end subroutine
function fun1()
  integer :: fun1
  fun1=1
end function
```

FLOOR Function

Description

The FLOOR function returns the greatest INTEGER number less than or equal to a REAL argument.

Syntax

FLOOR (*a [, kind]*)

Required Arguments

a is an INTENT(IN) scalar or array of type REAL.

Optional Arguments

kind is INTENT(IN) and determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *"KIND Function"* or *"SELECTED_INT_KIND Function"*.

Result

The result is an INTEGER number whose value is the largest integer less than or equal to *a*.

If *kind* is present, it specifies the kind of the result.

If *kind* is absent, the result is type default INTEGER.

Example

```
real :: r=4.7,x(3)=(/-.5,0.,.5/)
write(*,*) floor(r)                       ! writes 4
write(*,*) floor(-r,selected_int_kind(2)) ! writes -5
write(*,*) floor(x)                       ! writes -1 0 0
```

FLUSH Subroutine

Description

The FLUSH subroutine causes data stored in an output buffer to be written to its i/o unit, and clears the buffer.

Syntax

FLUSH (*iunit*)

Arguments

iunit is an INTENT(IN) scalar of type INTEGER. It is the unit number of the file whose buffer is to be written.

Remarks

Execution of the FLUSH subroutine does not flush the file buffer.

Example

```
call flush(11)  ! empty buffer for unit 11
```

FORALL Construct

Description

The FORALL construct controls execution of a block of assignment and pointer assignment statements. Execution in the block is selected by sets of index values and an optional mask expression.

Syntax

[construct-name:*]* FORALL (*forall-triplets [*, *mask]*)

[forall-body]

END FORALL *[construct-name]*

Where:

construct-name is an optional name for the FORALL construct.

forall-triplets 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 variable, which is an array index. *subscript* may not refer to an *index-name* in the same *forall-triplets* list.

stride is a scalar INTEGER variable, which is the array stride. *stride* may not make reference to an *index-name* in the same *forall-triplets* list.

mask is a scalar expression of type LOGICAL.

forall-body is zero or more assignment or pointer assignment statements, WHERE statements or constructs, or FORALL statements or constructs.

Remarks

Execution of a FORALL construct causes the set of values for *index-name* to be determined, and *mask* to be evaluated. Values for *index-name* are determined by taking the starting *subscript*, and incrementing it by the *stride* until a value falls outside the range *subscript : subscript*. *mask* is evaluated for each combination of *index-name* values, and assignments in the *forall-body* are made for those combinations of *index-names* for which *mask* evaluates to true.

If the FORALL construct has a *construct-name*, the same *construct-name* must appear at the beginning and end of the construct.

Any procedure referenced in *mask* or in *forall-body* must be a *"PURE Procedure"*.

If *mask* is not present it is as if it were present with the value true.

The set of values for *index-name* may be determined in any order.

The value of an *index-name* cannot be altered within the *forall-body*.

Example

```
real :: a(3,3)
integer :: i,j
forall(i=1:3,j=1:3:2)      ! selection by index values
  a(i,j)=real(i+j)
end forall
write(*,'(3(1x,f10.6))') a ! row 2 is all zeros
forall(i=1:3, j=1:3,a(i,j) == 0.) ! selection by index
                                  ! values and scalar mask
  a(i,j)=i*j ! this assignment is only done for
             ! elements of a that equal zero
end forall
write(*,'(3(1x,f10.6))') a
```

FORALL Statement

Description

The FORALL statement controls execution of an assignment or pointer assignment statement with selection by sets of index values and an optional mask expression.

Syntax

FORALL (*forall-triplets [, mask]*) *forall-assignment-stmt*

Where:

forall-triplets 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 variable, which is an array index. *subscript* may not refer to an *index-name* in the same *forall-triplets* list.

stride is a scalar INTEGER variable, which is the array stride. *stride* may not make reference to an *index-name* in the same *forall-triplets* list.

mask is a scalar expression of type LOGICAL.

forall-assignment-stmt is an assignment statement or a pointer assignment statement.

Remarks

Execution of a FORALL statement causes the set of values for *index-name* to be determined, and *mask* to be evaluated. Values for *index-name* are determined by taking the starting *subscript*, and incrementing it by the *stride* until a value falls outside the range *subscript : subscript*. *mask* is evaluated for each combination of *index-name* values, and assignments in the *forall-assignment-stmt* are made for those combinations of *index-names* for which *mask* evaluates to true.

Any procedure referenced in *mask* or in *forall-assignment-stmt* must be a *"PURE Procedure"*.

If *mask* is not present it is as if it were present with the value true.

The set of values for *index-name* may be determined in any order.

The value of an *index-name* cannot be altered within the *forall-assignment-stmt.*

Example

```
integer :: a(3,3)=(/1,2,3,4,5,6,7,8,9/),i,j
forall(i=1:3,j=1:3, j > i) a(i,j)=a(j,i)
! assigns the transpose of the lower triangle of array a
write(*,'(3(1x,f10.6))') a
```

FORMAT Statement

Description

The FORMAT statement provides explicit information that directs how data and characters are read on input and displayed on output.

Syntax

FORMAT (*[format-items]*)

Where:

format-items is a comma-separated list of *[r]data-edit-descriptor*, *control-edit-descriptor*, or *char-string-edit-descriptor*, or *[r]*(*format-items*)

data-edit-descriptor is I*w[.m]*
or B*w[.m]*
or O*w[.m]*
or Z*w[.m]*
or F*w.d*
or D*w.d*
or E*w.d[*E*e]*
or EN*w.d[*E*e]*
or ES*w.d[*E*e]*
or G*w.d[*E*e]*
or L*w*
or A*[w]*
w, *m*, *d*, and *e* are INTEGER literal constants that represent field width, digits, digits after the decimal point, and exponent digits, respectively.

control-edit-descriptor is T*n*
or TL*n*
or TR*n*
or *n*X
or S
or SP
or SS
or BN
or BZ
or *[r]*/
or :
or *k*P

char-string-edit-descriptor is a CHARACTER literal constant

rep-chars is a string of characters

c is the number of characters in *rep-chars*

r, *k*, and *n* are positive INTEGER literal constants used to specify a number of repetitions of the *data-edit-descriptor*, *char-string-edit-descriptor*, *control-edit-descriptor*, or (*format-items*)

Table 9: Format edit descriptors

Edit Descriptor	Interpretation	Intrinsic type
I*w[.m]*	ordinal number with field width of *w*, displays *m* digits	INTEGER
B*w[.m]*	binary number with field width of *w*, displays *m* digits	INTEGER
O*w[.m]*	octal number with field width of *w*, displays *m* digits	INTEGER
Z*w[.m]*	hexadecimal number with field width of *w*, displays *m* digits	INTEGER
F*w.d*	decimal number with field width of *w*, displays *d* decimal places, no exponent	REAL or COMPLEX
E*w.d[*E*e]* and D*w.d[*E*e]*	decimal number with field width of *w*, displays *d* decimal places, and an exponent with *e* digits	REAL or COMPLEX
EN*w.d[*E*e]*	decimal number with field width of *w*, displays *d* decimal places, and an exponent with *e* digits (engineering notation)	REAL or COMPLEX
ES*w.d[*E*e]*	decimal number with field width of *w*, displays *d* decimal places, and an exponent with *e* digits (scientific notation)	REAL or COMPLEX
G*w.d[*E*e]*	(generalized) field width of *w*, displays *d* decimal places, and an exponent with *e* digits	Any intrinsic type
L*w*	T or F with a field width of *w*	LOGICAL
A*[w]*	alphanumeric with a field width of *w*	CHARACTER
T*n*	move *n* spaces from the start of the record	None
TL*n*	move *n* spaces left of current position	None
TR*n*	move *n* spaces right of current position	None
*n*X	move *n* spaces right of current position	None
S	default generation of plus sign on subsequent output	Numeric

Table 9: Format edit descriptors

Edit Descriptor	Interpretation	Intrinsic type
SP	force generation of plus sign on subsequent output	Numeric
SS	no generation of plus sign for subsequent output	Numeric
BN	ignore non-leading blanks on input of subsequent items	Numeric
BZ	interpret non-leading blanks as zeros on input of subsequent items	Numeric
[r] /	skip to the next record *r* is a repeat count	None
:	terminates format control if there are no more items in the i/o list	None
*k*P	set a scale factor of *k* for subsequent items	REAL or COMPLEX

Remarks

The FORMAT statement must be labeled.

Edit descriptors may be nested within parentheses and may be preceded by a repeat factor. A parenthesized list of edit descriptors may also be preceded by a repeat factor, indicating that the entire list is to be repeated.

The comma between edit descriptors may be omitted in the following cases:

- between the scale factor (P) and the numeric edit descriptors F, E, EN, ES, D, or G
- before a new record indicated by a slash when there is no repeat factor present
- after the slash for a new record
- before or after the colon edit descriptor

Within a CHARACTER literal constant, if a string delimiter character (either an apostrophe or quote) is to appear as a part of the string, it must appear as a consecutive pair of the delimiter characters without any blanks. Each such pair represents a single occurrence of the delimiter character.

Example 1

```
!  numeric output editing
    integer :: i=-1
    real :: r=1.
    write(*,101) i ! writes        -1
    write(*,102) i ! writes     -0001
    write(*,103) i ! writes 11111111111111111111111111111111
    write(*,104) i ! writes 37777777777
    write(*,105) i ! writes  FFFFFFFF
    write(*,201) r ! writes   1000.00
    write(*,202) r ! writes  0.01D+02
    write(*,203) r ! writes  +0.10E+1
    write(*,204) r ! writes  1.00E+00
    write(*,205) r ! writes  1.00E+00
101 format(I10)    ! Show up to 10 digits, field width 10
102 format(I10.4)  ! Always show 4 digits, field width 10
103 format(B34.32) ! Show 32 binary digits, field width 34
104 format(O13.11) ! Show 11 octal digits, field width 13
105 format(Z10.8)  ! Show 8 hex digits, field width 10
201 format(3PF10.2)! 2 dec places field width 10 scale 3
202 format(-1P,D10.2) ! 2 dec places field width 10 scale -1
203 format(SP,E10.2E1)! 2 dec places, field width 10,
                      ! 1 digit exponent, produce plus sign
204 format(SSEN10.2E2)! 2 decimal places, field width 10,
                      ! 2 digit exponent suppress plus sign
205 format(ES10.2E2)  ! 2 decimal places, field width 10,
                      ! 2 digit exponent
```

Example 2

```
!  numeric input editing
    character(len=5) :: in_data1="11000"     ! internal file
    character(len=10) :: in_data2="    1    1"! internal file
    integer :: i
    real :: r
    complex :: q
    read(in_data1,101) i
    write(*,*) i         ! writes 1100000000
    read(in_data1,102) i
    write(*,*) i         ! writes 11000
    read(in_data1,103) i
    write(*,*) i         ! writes 24
    read(in_data1,104) i
    write(*,*) i         ! writes 4806
    read(in_data1,105) i
    write(*,*) i         ! writes 69632
    read(in_data1,201) r
    write(*,*) r         ! writes 110.
    read(in_data1,202) r
    write(*,*) r         ! writes 11000000.
    read(in_data2,202) r
    write(*,*) r         ! writes 11000.
    read(in_data2,203) r
    write(*,*) r         ! writes 100001.
    read(in_data2,204) q
    write(*,*) q         ! writes (1.,1.)
    read(in_data2,205) q
    write(*,*) q         ! writes (10.,100.)
101 format(BZI10) ! Interpret non leading blanks as zeros
102 format(BNI10) ! Ignore non leading blanks
103 format(B32)   ! Read up to 32 binary digits
104 format(O11)   ! Read up to 11 octal digits
105 format(Z8)    ! Read up to 8 hexadecimal digits
201 format(F10.2) ! last two digits are right of decimal
202 format(-3PF10.0)  ! Scale factor -3
203 format(BZF10.0)   ! non leading blanks are zeros
204 format(2(F6.0))   ! Ignore blanks
205 format(BZ,2(F6.0))! non leading blanks are zeros
```

Example 3

```
!  generalized, logical and character editing
    integer :: i
    real :: r
    real(kind(1.d0)) :: d
    complex :: q
    logical :: l
    character(len=10) :: rdstr(2)
    character(len=10) :: in_data="    1    1"
    character(len=20) :: in_str=" Howdy There, Folks!"
    read(in_data,301) i
    write(*,301) i           ! writes      11
    read(in_data,301) r
    write(*,301) r           ! writes 0.11
    read(in_data,301) d
    write(*,301) d           ! writes 0.11
    read(in_data,301) q
    write(*,301) q           ! writes 0.11    0.0
    read(in_str(8:8),301) l
    write(*,301) l           ! writes       T
    read(in_str(15:15),301) l
    write(*,301) l           ! writes       F
    read(in_str,301) rdstr
    write(*,301) rdstr       ! writes Howdy There, Folks!
    read(in_str(8:8),401) l
    write(*,401) l           ! writes       T
    read(in_str(15:15),401) l
    write(*,401) l           ! writes       F
    read(in_str,501) rdstr
    write(*,501) rdstr        ! writes Howdy There, Folks!
    write(*,501) "howdy"      ! writes   howdy
    write(*,501) '"howdy"'    ! writes "howdy"
    write(*,501) "'howdy'"    ! writes 'howdy'
    write(*,501) """howdy"""  ! writes "howdy"
    write(*,501) '''howdy'''  ! writes 'howdy'
301 format(2G10.2) ! general editing, field width 10
401 format(L10)    ! Logical T or F, field width 10
501 format(2A10)   ! Alphanumeric string, field width 10
```

Example 4

```
!  positional editing
    real :: r(3)=(/-1., 0., 1./)
    write(*,201) r ! writes   -1.00   0.00   1.00
    write(*,202) r ! writes    1.00  -1.00   0.00
    write(*,203) r ! writes    1.00   0.00  -1.00
    write(*,204) r ! writes   -1.00   0.00   1.00
    write(*,205) r ! writes          -1.00
                   !                  0.00
                   !                  1.00
201 format(TR10,3F10.2)
202 format(T21,F10.2,T31,F10.2,T11,F10.2)
203 format(TR30,F10.2,2(TL20,F10.2))
204 format(10X,3(F10.2))
205 format(3(T21,F10.2,/))
```

Example 5

```
! formats without statements
integer :: i_a(3,3)=reshape((/1,2,3,4,5,6,7,8,9/), &
                            shape(i_a))
integer :: i_b(2,3)=reshape((/1,2,3,4,5,6/),shape(i_b))
integer :: i_c(3,2)=reshape((/1,2,3,4,5,6/),shape(i_c))
call write_array2d(i_a)
call write_array2d(i_b)
call write_array2d(i_c)
contains
  subroutine write_array2d(i) ! compose a format and
    integer :: i(:,:)         ! write rank two array
    character(40) :: fmt      ! in rows and columns
    write(fmt,*) "(",size(i,1),"(",size(i,2),"I10,/))"
    write(*,*) fmt ! write the format string
    write(*,fmt) i ! write array using the format string
  end subroutine
end program
```

FRACTION Function

Description
The Fraction function returns the fractional part of the representation of a REAL number.

Syntax
FRACTION (*x*)

Arguments
x is an INTENT(IN) scalar or array of type REAL.

Result
The result is of type REAL and the same kind as *x*. Its value is the fraction part of the physical representation of *x*.

Example
```
real :: x=4.3
write(*,*) x,fraction(x)
           ! writes 4.300000 0.5375000
write(*,*) scale(fraction(x),exponent(x))
           ! writes 4.300000
```

FUNCTION Statement

Description
The FUNCTION statement begins a function subprogram. It specifies the functions name and dummy arguments, and any special characteristics such as PURE, ELEMENTAL, or RECURSIVE. It may optionally specify the functions' return type, and the name of a result variable used within the subprogram to assign a value to the function result.

Syntax
*[*PURE*][*ELEMENTAL*][*RECURSIVE*] [type-spec]* FUNCTION *function-name* (*[dummy-arg-names]*) *[*RESULT *(result-name*)*]*

Where:
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 TYPE (*type-name*)

kind-selector is (*[*KIND=*] kind*)

char-selector is (*[*LEN=*] length [*, KIND=*kind]*)
or (KIND=*kind [*, LEN=*length]*)
or * *char-length [,]*

kind is a scalar INTEGER expression that can be evaluated at compile time

length is a scalar INTEGER expression
or *

char-length is a scalar INTEGER literal constant
or (*)

function-name is the name of the function

dummy-arg-names is a comma-separated list of dummy argument names

result-name is the name of the result variable

Remarks

A function with the prefix PURE or ELEMENTAL is subject to the additional constraints of pure procedures, which ensure that no unseen side effects occur on invocation of the function. See "PURE Procedure" on page 225.

An ELEMENTAL function is subject to the constraints of elemental procedures. See "ELEMENTAL Procedure" on page 123.

A function cannot be both ELEMENTAL and RECURSIVE.

The keyword RECURSIVE must be present if any function defined by a FUNCTION or ENTRY statement within the subprogram directly or indirectly calls itself.

A RECURSIVE function that calls itself directly must use the RESULT option.

If RESULT is omitted, then *function-name* is the result variable.

If the function result is an array or a pointer, this must be specified in the declaration of the result variable in the function body.

Example 1

```
!   basic function declaration
function func1(a,b)
  real :: func1 ! result type defined here
  real,intent(in) :: a,b
  func1=a-b ! function is assigned a result
end function
!   function declaration with result variable
function func2(a,b) result(res)
  real :: res   ! result type defined here
  real,intent(in) :: a, b
  res=a-b   ! function is assigned a result
end function
!   function declaration with type
real function func3(a,b) ! result type defined here
  real, intent(in) :: a,b
  func3=a-b          ! function is assigned a result
end function
!   program invoking functions
  write(*,*) func1(-1.,1.) ! writes -2.
  write(*,*) func2(-1.,1.) ! writes -2.
  write(*,*) func3(-1.,1.) ! writes -2.
end
```

Example 2

```
!   recursive function with result variable
recursive function func4(a,b) result(res)
  real :: res     ! result type defined here
  real :: a,b
  if (a >= b) then
      res=a-b   ! function returns
  else if (a < spacing(b)) then
    res=-b        ! function returns
  else if (a < 0.) then
    b=func5(-a,b) ! indirect recursion
  else
    a=func4(a,-b) ! direct recursion
  end if
end function
!   recursive function without result variable
recursive function func5(a,b)
  real :: func5      ! result type defined here
  real :: a,b
  if (a < b) then
    func5=func4(b,a) ! result variable not required
  else                 ! if recursive function does
    func5=func4(a,b) ! not invoke itself directly
  end if
end function
!   program invoking functions
  write(*,*) func4(-1.,1.),func4(1.,-1.) ! writes -1. 2.
  write(*,*) func5(-1.,1.),func5(1.,-1.) ! writes  2. 2.
end
```

GETCL Subroutine

Description

The GETCL subroutine gets command line arguments.

Syntax

GETCL (*cl_args*)

Arguments

cl_args is an INTENT(OUT) scalar of type CHARACTER. It contains any text which is provided when the program is invoked beginning with the first non-white-space character after the program name.

Remarks

If the length of *cl_args* is not sufficient to hold the entire command line argument, the left-most characters in the argument are retained up to the length of the character variable

If any run-time options are present (-Wl,...), they are returned by GETCL.

If no command line argument is present, *cl_args* contains blank characters.

Example

```
character(256) :: cl_arg1
call getcl(cl_arg1)
write(*,*) len_trim(cl_arg1),trim(cl_arg1)
```

GETENV Subroutine

Description

The GETENV subroutine gets the value of the specified environment variable.

Syntax

GETENV(*env_var*, *env_value*)

Arguments

env_var is an INTENT(IN) scalar of type CHARACTER. It specifies the environment variable whose value is requested.

env_value is an INTENT(OUT) scalar of type CHARACTER. On return, it contains the value of the environment variable *env_var*.

Remarks

If the specified environment variable does not exist, on return, *env_value* contains null characters (CHAR(0)).

If the length of *env_value* is not sufficient to hold the entire environment value, the leftmost characters are retained up to the length of the character variable

Example

```
character(4096) :: env_value1
call getenv("PATH",env_value1)
write(*,*) len_trim(env_value1),trim(env_value1)
```

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 must be the label of a branch target statement in the same scoping unit as the GOTO statement.

Example

```
      a=b
      go to 10 ! branches to 10
      b=c      ! never executed
10    c=d
```

HUGE Function

Description

The HUGE function returns the largest representable number of the argument’s data type.

Syntax

HUGE (x)

Arguments

x is an INTENT(IN) scalar or array of type REAL or INTEGER.

Result

The result is of the same type and kind as *x*. Its value is the value of the largest number representable by the data type of *x*.

Example

```
real(kind(1.e0)) :: r10
real(kind(1.d0)) :: r100
real(kind(1.q0)) :: r1000
integer(selected_int_kind(r=1)) :: i1
integer(selected_int_kind(r=4)) :: i4
integer(selected_int_kind(r=7)) :: i7
integer(selected_int_kind(r=12)) :: i12
write(*,*) huge(r10)   ! writes 3.40282347e+38
write(*,*) huge(r100)  ! writes 1.797693134862316e+308
write(*,*) huge(r1000) ! writes 1.18973....28007e+4932
write(*,*) huge(i1)    ! writes 127
write(*,*) huge(i4)    ! writes 32767
write(*,*) huge(i7)    ! writes 2147483647
write(*,*) huge(i12)   ! writes 9223372036854775807
```

IACHAR Function

Description

The IACHAR function returns the position of a character in the ASCII collating sequence. See “ASCII Character Set” on page 319.

Syntax

IACHAR (*c*)

Arguments

c is an INTENT(IN) scalar or array of type CHARACTER. *c* must have a length of one.

Result

The result is of type default INTEGER. Its value is the position of *c* in the ASCII collating sequence. It is in the range $0 \le iachar(c) \le 127$.

If *c* is an array, the result is an array of integer values with the same shape as *c*.

Example

```
character(len=1) :: c1='A',c3(3)=(/"a","b","c"/)
```

```
write(*,*) iachar(c1) ! writes   65
write(*,*) iachar(c3) ! writes   97 98 99
```

IAND Function

Description

The IAND function performs a bit-wise logical AND operation on two integer arguments.

Syntax

IAND (*i*, *j*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

j is an INTENT(IN) INTEGER of the same kind as *i*. If *i* is an array, *j* must have the same shape as *i*.

Result

The result is of type INTEGER. Its value is the result of performing a bit-wise logical AND operation on *i* and *j*.

Example

```
i=53         ! i=00110101 binary (lowest order byte)
j=45         ! j=00101101 binary (lowest order byte)
k=iand(i,j)  ! k=00100101 binary (lowest order byte)
             ! k=37 decimal
```

IBCLR Function

Description

The IBCLR function sets a single bit in an integer argument to zero.

Syntax

IBCLR (*i*, *pos*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

pos is an INTENT(IN) scalar or array of type INTEGER. The value of *pos* must be within the range zero to (BIT_SIZE(i)-1). See “BIT_SIZE Function” on page 81.

If *i* and *pos* are both arrays, they must have the same shape.

Result

The result is of type default INTEGER. Its value is *i* with the bit at position *pos* is set to zero.

If *i* is an array and *pos* is scalar, the result is an array with the same shape as *i*. Each element of the resulting array has the bit at position *pos* set to zero.

If *i* is scalar and *pos* is an array, the result is an array with the same shape as *pos*. Each element of the resulting array contains the value of *i* with the bit indicated by the corresponding element of *pos* set to zero.

If *i* and *pos* are both arrays, the result is an array with the same shape as *i*. Each element of the resulting array contains the value from the corresponding element of *i* with the bit indicated by the corresponding element of *pos* set to zero.

Example

```
integer :: i=-1,p=3,ia(2)=(/-1,7/),pa(2)=(/1,2/)
write(*,"(b34)") i             ! writes     0
write(*,"(b34)") ibclr(i,p)    ! writes  1000
write(*,"(2b34)") ibclr(i,pa)  ! writes    10      100
write(*,"(2b34)") ia           ! writes 111...111111
write(*,"(2b34)") ibclr(ia,p)  ! writes 111...110111
write(*,"(2b34)") ibclr(ia,pa) ! writes 111...111101
```

IBITS Function

Description

The IBITS function extracts a sequence of bits from an integer argument.

Syntax

IBITS (*i*, *pos*, *len*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

pos is an INTENT(IN) scalar or array of type INTEGER. It must be non-negative.

len is an INTENT(IN) scalar or array of type INTEGER. It must be non-negative and *pos*+*len* must be less than or equal to BIT_SIZE(*i*). See “BIT_SIZE Function” on page 81.

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.

If any argument is an array, the result is an array and has the same shape as the argument array. The value of each element is the value of the scalar operation performed on corresponding elements of any array arguments.

Note that the lowest order position starts at zero.

Example

```
integer :: i; data i/z'0f0f'/
write(*,"(b34)") i            ! writes 111100001111
write(*,"(b34)") ibits(i,0,4) ! writes         1111
write(*,"(b34)") ibits(i,4,5) ! writes        10000
```

IBSET Function

Description

The IBSET function sets a single bit to one.

Syntax

IBSET (*i*, *pos*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

pos is an INTENT(IN) scalar or array of type INTEGER. The value of *pos* must be within the range zero to (BIT_SIZE(i)-1). See “BIT_SIZE Function” on page 81.

If *i* and *pos* are both arrays, they must have the same shape.

Result

The result is of type INTEGER and of the same kind as *i*. Its value is *i* with the bit at position *pos* is set to one.

If *i* is an array and *pos* is scalar, the result is an array with the same shape as *i*. Each element of the resulting array has the bit at position *pos* set to one.

If *i* is scalar and *pos* is an array, the result is an array with the same shape as *pos*. Each element of the resulting array contains the value of *i* with the bit indicated by the corresponding element of *pos* set to one.

If *i* and *pos* are both arrays, the result is an array with the same shape as *i*. Each element of the resulting array contains the value from the corresponding element of *i* with the bit indicated by the corresponding element of *pos* set to one.

Example

```
integer :: i=0,p=3,ia(2)=(/0,0/),pa(2)=(/1,2/)
```

```
write(*,"(b34)") i            ! writes      0
write(*,"(b34)") ibset(i,p)    ! writes   1000
write(*,"(2b34)") ibset(i,pa)  ! writes     10       100
write(*,"(2b34)") ia           ! writes      0         0
write(*,"(2b34)") ibset(ia,p)  ! writes   1000      1000
write(*,"(2b34)") ibset(ia,pa) ! writes     10       100
```

ICHAR Function

Description

The ICHAR function returns the position of a character in the collating sequence associated with the kind of the character. The only character set supported is the ASCII character set, with a kind number of 1, containing 127 characters. See “ASCII Character Set” on page 319.

Syntax

ICHAR (*c*)

Arguments

c is an INTENT(IN) scalar or array of type CHARACTER with a length of one.

Result

The result is of type default INTEGER. Its value is the position of *c* in the collating sequence associated with the kind of *c* and is in the range $0 \le ichar(c) \le n - 1$, where *n* is the number of characters in the collating sequence.

Example

```
character(len=1) :: c(6)=(/"H","o","w","d","y","!"/)
write(*,*) ichar(c)  ! writes 72 111 119 100 121 33
```

IEOR Function

Description

The IEOR function performs a bit-wise logical exclusive OR operation on two integer arguments.

Syntax

IEOR (*i*, *j*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

j is an INTENT(IN) scalar or array of type INTEGER and is the same kind as *i*.

Result

The result is of type INTEGER. Its value is obtained by performing a bit-wise logical exclusive OR operation on *i* and *j*.

Example

```
i=53         ! i=00110101 binary (lowest order byte)
j=45         ! j=00101101 binary (lowest order byte)
k=ieor(i,j) ! k=00011000 binary (lowest order byte)
             ! k=24 decimal
```

IF Construct

Description

The IF construct controls whether a block of statements or executable constructs will be executed based on the value of a logical expression.

The IF-THEN statement signals the beginning of an IF construct.

The ELSE IF statement controls execution of a block of code where all previous IF expressions in the construct were false.

The ELSE statement controls execution of a block of code where all other IF expressions in the construct were false.

The END IF statement signals the end of the innermost nested IF construct.

Syntax

[construct-name:*]* IF (*expr*) THEN
        block
[ELSE IF (*expr*) THEN *[construct-name]*
        block]
...
[ELSE *[construct-name]*
        block]
END IF *[construct-name]*

Where:

construct-name is an optional name for the construct.

expr is a scalar LOGICAL expression.

block is a sequence of zero or more statements or executable constructs.

Remarks

The *expr*s 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 is found, the block immediately following is executed and this completes the execution of the construct. The *expr*s in any remaining ELSE IF statements of the IF construct are not evaluated.

If none of the evaluated expressions is true, then the block of code following the ELSE statement is executed. If there is no ELSE statement, the execution of the construct is completed without the execution of any block within the construct.

The ELSE statement and its block of code must be the last block to appear in the IF construct.

If the IF statement specifies a construct name, the corresponding END IF statement must specify the same construct name. If the IF statement does not specify a construct name, the corresponding END IF statement must not specify a construct name.

Example 1

```
integer :: i=0
if (i > 10) then
  write(*,*) "i is greater than ten"
else if (i > 0) then
  write(*,*) "i is less than ten but greater than zero"
else if (i < 0) then
  write(*,*) "i is less than zero"
else
  write(*,*) "i equals zero"
end if
```

Example 2

```
   logical :: exp1=.true.,exp2=.false.
outer_if: &
   if (exp1) then
inner_if: &
      if(exp2) then
      end if inner_if
   end if outer_if
   if(exp1 .eqv. exp2) then
   end if
```

IF Statement

Description

The IF statement controls whether or not a statement is executed based on the value of a logical expression.

Syntax

IF (*expr*) *action-statement*

Where:

expr is a scalar LOGICAL expression.

action-statement is an executable statement other than another IF or the END statement of a program, function, or subroutine.

Remarks

Execution of an IF statement causes evaluation of the logical expression.

If the expression is true, the *action-statement* is executed.

If the value is false, the *action-statement* is not executed.

Example

```
real :: a=-1
if (a < 0) write(*,*) " a must be less than zero,&
           & because this statement was executed"
```

IMPLICIT Statement

Description

The IMPLICIT statement specifies a type and optionally a kind or a CHARACTER length for each variable or function name beginning with the letter(s) specified in the IMPLICIT statement. Alternately, it can specify that no implicit typing is to apply in the scoping unit.

Syntax

IMPLICIT *implicit-specs*

or

IMPLICIT NONE

Where:

implicit-specs is a comma-separated list of *type-spec* (*letter-specs*)

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 TYPE (*type-name*)

kind-selector is (*[*KIND *=] kind*)

char-selector is (*[*LEN=*] length [*, KIND=*kind]*)
or (KIND=*kind [*, LEN=*length]*)
or * *char-length [,]*

type-name is the name of a user-defined type.

kind is a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *"KIND Function"*, *"SELECTED_INT_KIND Function"*, or a *"SELECTED_REAL_KIND Function"*, as appropriate.

length is a scalar INTEGER expression
or *

char-length is a scalar INTEGER literal constant
or (*)

letter-specs is a comma-separated list of *letter[-letter]*

letter is one of the letters A-Z.

Remarks

Any data entity that is not explicitly declared by a type or function declaration statement, is not an intrinsic function, and is not made accessible by host or use association, is declared implicitly to be of the type (and type parameters, kind and length) mapped from the first letter of its name.

Implicit typing for a range of letters can be specified by separating the beginning letter in the range and the ending letter in the range by a hyphen character. This is equivalent to writing a list containing all of the letters in alphabetical order in the alphabetic sequence from the beginning letter through the ending letter.

The same letter must 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.

In the absence of an implicit statement, a program unit is treated as if it had a host with the declaration

```
      implicit integer (i-n), real (a-h, o-z)
```

IMPLICIT NONE specifies that no implicit typing will occur, and all data entities that are local in scope or appear in a common block within the scoping unit must be declared in a type declaration statement.

If IMPLICIT NONE is specified in a scoping unit, it must precede any other specification statements that appear, and the scoping unit cannot contain any other IMPLICIT statements.

Example 1

```
      implicit type(vowel) (a,e,i,o,u) ! if a variable name does
      implicit character (c)           ! not appear in a type
```

```
implicit integer (j,k,m,n)       ! declaration statement,
implicit logical (l)             ! the type is determined
implicit real (b, d, f-h, p-t, v-z) ! by the first letter of
type vowel                       ! the variable name, as
  character :: v                 ! indicated by these
end type vowel                   ! implicit statements
```

Example 2

```
implicit none ! requires explicit type declarations for
              ! each local variable
integer :: ill
il1=0 ! implicit none will allow the compiler to
      ! catch typos. this statement will generate
      ! a compiler error because the number one was
      ! used instead of letter "l"
ill=O ! this typo will also be detected. the letter
      ! "O" was used instead of the number zero
```

INCLUDE Line

Description

The INCLUDE line causes text in a separate file to be processed as if the text replaced the INCLUDE line. The INCLUDE line is not a Fortran statement.

Syntax

INCLUDE *filespec*

Where:

filespec is a CHARACTER literal constant that corresponds to a path and file that contains source text to replace the INCLUDE line.

Remarks

The INCLUDE line must 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 processed by the compiler, not by a preprocessor.

Example

```
include "types.for"  ! include a file named types.for
                     ! in place of this INCLUDE line
```

INDEX Function

Description

The INDEX function returns the starting position of a substring within a string.

Syntax

INDEX (*string*, *substring [*, *back]*)

Required Arguments

string is an INTENT(IN) scalar or array of type CHARACTER.

substring is an INTENT(IN) scalar or array of type CHARACTER.

Optional Arguments

back is an INTENT(IN) scalar of type LOGICAL.

Result

The result is of type default INTEGER.

If *back* is absent or false, the result value is the position in *string* where the first instance of *substring* begins.

If *back* is true, the result value is the position number in *string* where the last instance of *substring* begins.

If *substring* is not found, or if *string* is shorter than *substring*, the result is zero.

If *substring* is of zero length, and *back* is absent or false, the result value is one.

If *substring* is of zero length, and *back* is true, the result value is LEN(*string*)+1.

Example

```
character(len=20) :: c1   =  "Howdy There!        ", &
                     c2(3)=(/"To be or not to be  ", &
                             "Believe it or not   ", &
                             "I'll be there       "/)
character(len=2) :: s2(3)=(/"be", "Be", "ow"/)
write(*,*) index(c1,"The")           ! writes  7
write(*,*) index(c1,s2)              ! writes  0  0  2
write(*,*) index(c2,"be")            ! writes  4  0  6
write(*,*) index(c2,s2, back=.true.) ! writes 17  1  0
```

INQUIRE Statement

Description

The INQUIRE statement enables a program to make inquiries about a unit or file's existence, connection, access method or other properties.

Syntax

INQUIRE *(inquire-specs*)

or

INQUIRE (IOLENGTH=*iolength*) *output-items*

Where:

inquire-specs is a comma-separated list of
*[*UNIT =*] external-file-unit*
or FILE=*file-name-expr*
or IOSTAT=*iostat*
or 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 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 FLEN=*flen*
or BLOCKSIZE=*blocksize*
or CONVERT =*file-format*
or CARRIAGECONTROL=*carriagecontrol*

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.

iostat is a scalar default INTEGER variable that is assigned a positive value if an error condition occurs, a negative value if an end-of-file or end-of-record condition occurs, and zero otherwise.

label is the statement label of the statement branched to if an error occurs.

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 default INTEGER variable that is assigned the value of the input/output unit of the external file or -1 if the file is not connected or does not exist.

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 evaluates to SEQUENTIAL if the file is connected for sequential access, DIRECT if the file is connected for direct access, TRANSPARENT if the file is connected for transparent 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 or does not exist.

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 or does not exist.

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, 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 or does not exist.

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 or does not exist.

recl is a scalar default INTEGER variable that evaluates 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, or zero if the file is not connected or does not exist.

nextrec is a scalar default 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, the value becomes zero.

blank is a scalar default CHARACTER variable that evaluates to 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 or does not exist.

position is a scalar default CHARACTER variable that evaluates to REWIND if the newly opened sequential access file is positioned at its initial point; APPEND if it is positioned before the endfile record if one exists and at the file terminal point otherwise; ASIS if the position is after the endfile record; and UNDEFINED if the file is not connected or does not exist.

action is a scalar default CHARACTER variable that evaluates to 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 or does not exist.

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 or does not exist.

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 or does not exist.

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 or does not exist.

delim is a scalar default CHARACTER variable that evaluates to APOSTROPHE if the apostrophe is used to delimit character constants written with list-directed or namelist formatting, QUOTE if the quotation mark is used, NONE if neither quotation marks nor apostrophes is used, and UNDEFINED if the file is not connected or does not exist.

pad is a scalar default CHARACTER variable that evaluates to YES if the formatted input record is padded with blanks or if the file is not connected or does not exist, and NO otherwise.

flen is a scalar default INTEGER variable that is assigned the length of the file in bytes.

blocksize is a scalar default INTEGER variable that evaluates to 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. The value is zero if the file is not connected or does not exist.

file-format is a scalar default CHARACTER variable that evaluates to BIG_ENDIAN if big endian conversion is in effect, LITTLE_ENDIAN if little endian conversion is in effect, IBM if IBM style conversion is in effect, and NATIVE if no conversion is in effect.

carriagecontrol is a scalar default CHARACTER variable that evaluates to FORTRAN if the first character of a formatted sequential record is used for carriage control, and LIST otherwise.

iolength is a scalar default INTEGER variable that is assigned a value that would result from the use of *output-items* in an unformatted output statement. The value is 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-items*.

output-items is a comma-separated list of items used with *iolength* as explained immediately above.

Remarks

When the INQUIRE statement is executed for a file or unit that is not connected, information about that file or unit is limited to the existence of the file, and the connection status of the file or unit.

inquire-specs must contain one FILE= specifier or one UNIT= specifier, but not both, and at most one of each of the other specifiers.

In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in *inquire-specs*.

When a returned value of a specifier other than the NAME= specifier is of type CHARACTER and the processor is capable of representing letters in both upper and lower case, the value returned is in upper case.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry specifier variables become undefined, except for the variable in the IOSTAT= specifier (if any).

Example

```
logical :: lopened,lexist
integer :: this_unit=10
inquire (this_unit,opened=lopened)
if(lopened) then
  write(*,*) " Unit ",this_unit," is open!"
else
  write(*,*) " Unit ",this_unit," is not open!"
end if
```

```
inquire (file="inquire.f90",exist=lexist)
if(lexist) then
  write(*,*) " The file 'inquire.f90' exists!"
else
  write(*,*) " The file 'inquire.f90' does not exist!"
end if
```

INT Function

Description

The INT function converts a numeric argument to the INTEGER type.

Syntax

INT (*a [*, *kind]*)

Required Arguments

a is an INTENT(IN) scalar or array of type INTEGER, REAL, or COMPLEX.

Optional Arguments

kind is INTENT(IN) and determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *"KIND Function"* or *"SELECTED_INT_KIND Function"*.

Result

The result is of type INTEGER. It is the value of *a* without its fractional part.

If *kind* is present, the kind is that specified by *kind*.

If *a* is of type COMPLEX, the result is the value of the real part of *a* without its fractional part.

Example

```
integer :: i2=selected_int_kind(2)
integer(selected_int_kind(4)) :: i4=3
real :: a=2.5
complex :: c=(1.5,2.5)
write(*,*) i4,int(i4,i2) ! converts between integer kinds
write(*,*) a,int(a)      ! converts real to integer
write(*,*) c,int(c)      ! converts complex to integer
```

INTEGER Statement

Description

The INTEGER statement declares entities having the INTEGER data type.

Syntax

INTEGER *[kind-selector] [[, attribute-list] ::] entity [, entity] ...*

Where:

kind-selector is (*[*KIND=*] scalar-int-initialization-expr*)

scalar-int-initialization-expr is a scalar INTEGER expression that can be evaluated at compile time.

attribute-list is a comma-separated list from the following attributes: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), EXTERNAL, INTENT(IN) or INTENT(OUT) or INTENT(IN OUT), PUBLIC or PRIVATE, INTRINSIC, OPTIONAL, POINTER, SAVE, TARGET, VOLATILE, DLL_EXPORT or DLL_IMPORT or ML_EXTERNAL.

entity is *entity-name [*(*array-spec*)*] [=initialization-expr]*
or *function-name [*(*array-spec*)*]*

array-spec is an array specification.

initialization-expr is an expression that can be evaluated at compile time.

entity-name is the name of the entity being declared.

function-name is the name of the function being declared.

Remarks

function-name must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

The =*initialization-expr* must appear if the statement contains a PARAMETER attribute.

If =*initialization-expr* appears, a double colon must appear before the list of *entities*. Each *entity* has the SAVE attribute, unless it is in a named common block.

The =*initialization-expr* must not appear if *entity-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 pointer, an external name, an intrinsic name, or an automatic object.

If an array or function with an array result is declared with the POINTER attribute, it must have a deferred shape.

If an array is declared with the ALLOCATABLE attribute, it must have a deferred shape.

If an array or function with an array result does not have the POINTER or the ALLOCATABLE attribute, it must be specified with an explicit shape.

If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attributes must not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attributes must not be specified.

The PARAMETER attribute cannot be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

The INTENT(IN), INTENT(OUT), INTENT(IN OUT), and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* may not have the PUBLIC attribute if its type has the PRIVATE attribute.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

An *entity* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

A subprogram must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

An *entity* having the ALLOCATABLE attribute cannot be a dummy argument or a function result.

An array must not have both the ALLOCATABLE attribute and the POINTER attribute.

If an *entity* has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

An *entity* may not be given the same attribute more than once in a scoping unit.

Example

```
integer :: a,b(2,4)          ! explicit shape
integer,dimension(2,2) :: c=reshape((/1,2,3,4/),shape(c))
                             ! has save attribute
integer,pointer :: d(:)      ! deferred shape
integer,allocatable :: e(:)
integer,parameter :: f=3     ! must be initialized
```

INTENT Statement

Description

The INTENT statement specifies the treatment dummy arguments.

Syntax

INTENT(*intent-spec*) *[::] dummy-args*

Where:

intent-spec is IN
or OUT
or IN OUT

dummy-args is a comma-separated list of dummy arguments.

Remarks

The INTENT(IN OUT) attribute specifies that the dummy argument is intended for use both to receive data from and to return data to the invoking procedure. This is the default behavior if no INTENT attribute is specified.

The INTENT(IN) attribute specifies that the dummy argument is intended to receive data from the invoking procedure. The dummy argument's value may not be altered during the execution of the subprogram.

The INTENT(OUT) attribute specifies that the dummy argument is intended to return data to the invoking procedure. The subprogram must provide a value for all INTENT(OUT) arguments. An INTENT(OUT) dummy variable cannot be referenced within the subprogram until it has been assigned a value. If a value is not supplied in the subprogram, the value of the actual argument will become undefined upon the subprograms return.

The INTENT statement must not contain a dummy argument that is a procedure or a pointer.

Example

```
subroutine ex (a,b,c,d,e,f)
  real :: a,b,c
  intent(in) :: a      ! a cannot be assigned a value
  intent(out) b        ! b must be given a value
  intent(in out) c     ! default behavior
  real,intent(in) :: d ! intent attributes
  real,intent(out) :: e
  real,intent(in out) :: f
end subroutine ex
```

INTERFACE Block

Description

An INTERFACE block specifies the forms of reference by which a procedure can be invoked. An interface block specifies a procedure interface, a defined operation, or a defined assignment.

An INTERFACE statement begins an interface block.

An END INTERFACE statement ends an interface block.

Syntax

INTERFACE *[generic-spec]*

interface-spec

END INTERFACE *[generic-spec]*

Where:

generic-spec is the name of a generic procedure
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

defined-operator is one of the intrinsic operators or *.operator-name*.
operator-name is a user-defined name for the operation, consisting of one to 31 letters.

interface-spec specifies whether the procedure is a subroutine or a function, and any dummy arguments that the procedure might have. If the interface is a generic interface, interface operator or interface assignment, *interface-spec* may also specify that the procedure is a module procedure, as long as that procedure appears in a module that is within the scope of the procedure declaring the interface, or is available by use association.

Remarks

Explicit interface

An explicit interface for a procedure consists of the characteristics of the procedure, the name of the procedure, and the name and characteristics of any dummy arguments.

A dummy argument name appearing in an interface specifies the keyword for that dummy argument.

Explicit interfaces cannot be specified for procedures that are contained within a MODULE, because the interfaces for module procedures are provided implicitly.

Generic interface

An interface statement with a *generic-name* specifies a generic interface for each of the procedures in the interface block. Each procedure in the interface block may have an explicit interface, or may name a module procedure that is contained in the same module.

Each procedure's argument list within the generic interface must have a calling sequence that is unique, otherwise the interface is ambiguous, and an error will be produced at compile time.

The specific procedure is selected at runtime, based on the argument list used when the generic procedure is called.

If the interface is a generic, assignment or operator interface, if a *generic-spec* is present in the END INTERFACE statement, it must be identical to the generic spec in the INTERFACE statement.

If the interface is not a generic, assignment or operator interface, *generic-spec* cannot be present in the END INTERFACE statement.

Defined operations

If OPERATOR is specified in an INTERFACE statement, all of the procedures specified in the interface block must be functions that can be referenced as defined operations. In the case of binary operators, the function requires two arguments. In the case of unary operators, a function with one argument must be used.

OPERATOR must not be specified for functions with no arguments or for functions with more than two arguments.

The dummy arguments must be non-optional data objects and must be specified with INTENT((IN).

The function result must not have assumed CHARACTER length.

If the operator is an intrinsic-operator, the number of function arguments must be consistent with the intrinsic uses of that operator.

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.).

Overriding an intrinsic operator for an intrinsic data type is not allowed. Operators may only be defined for data types which do not already have a definition for that particular operator.

Defined assignments

If ASSIGNMENT is specified in an INTERFACE statement, all the procedures in the interface block must be subroutines that can be referenced as defined assignments.

Each subroutine must have exactly two dummy arguments.

Each argument must be non-optional.

The first argument must have the INTENT(OUT) or INTENT(IN OUT) attribute.

The second argument must have the INTENT(IN) attribute.

A defined assignment is treated as a reference to the subroutine, with the first argument as the assignment target, and the second argument as the expression to be assigned.

The ASSIGNMENT generic specification specifies that the assignment operation is extended or redefined if both sides of the equals sign are of the same derived type.

Example 1

```
! explicit interfaces
interface
  subroutine ex(a,b,c)
    implicit none
    real :: a,b(10,2)
    integer :: c
  end subroutine ex
  function why(t,f)
    implicit none
    logical,intent(in) :: t,f
    logical :: why
  end function why
end interface
```

Example 2

```
! generic interfaces
module mod1
  interface swap
    module procedure complex_swap ! interface for a module
    module procedure logical_swap ! procedure is implicit
  end interface
contains
  subroutine complex_swap(cx,cy) ! implicit interface
    complex :: cx, cy, ct        ! is defined here
    ct=cx
    cx=cy
    cy=ct
  end subroutine
  subroutine logical_swap(lx,ly) ! implicit interface
    logical :: lx,ly,lt          ! is defined here
    lt=lx
    lx=ly
    ly=lt
  end subroutine
end module
subroutine real_swap(x,y)
  real :: x,y,t
  t=x
  x=y
  y=t
end subroutine
subroutine int_swap(ix,iy)
  integer :: ix,iy,it
  it=ix
  ix=iy
  iy=it
end subroutine
program interface2
  use mod1
  interface swap ! extends the interface defined in mod1
    subroutine real_swap(x,y)  ! explicit interface
      implicit none
      real, intent(in out) :: x,y
    end subroutine real_swap
    subroutine int_swap(ix,iy) ! explicit interface
      implicit none
      integer,intent(in out) :: ix,iy
    end subroutine int_swap
  end interface
  real :: a=1,b=2
  integer :: ia=1,ib=2
  complex :: ca=(1,1),cb=(2,2)
  logical :: la=.true.,lb=.false.
  call swap(a,b)                ! calls real_swap
  write(*,*) a,b
  call swap(ia,ib)              ! calls int_swap
  write(*,*) ia,ib
  call swap(ca,cb)              ! calls complex_swap
  write(*,*) ca,cb
  call swap(la,lb)              ! calls logical_swap
  write(*,*) la,lb
end program
```

Example 3

```
! operator interfaces
module mod1
  interface operator (+)           ! binary operator
    module procedure char_plus_int ! implicit interface
    module procedure int_plus_char ! implicit interface
  end interface operator (+)
  interface operator (.not.)       ! unary operator
    module procedure int_not       ! implicit interface
    function real_not(a)           ! explicit interface
      real :: real_not
      real,intent(in) :: a
    end function
  end interface operator (.not.)
contains
  function char_plus_int(c,i)
    character :: char_plus_int
    integer,intent(in) :: i
```

```
      character,intent(in) :: c
      char_plus_int=int_plus_char(i,c)
    end function char_plus_int
    function int_plus_char(i,c)
      character :: int_plus_char
      integer,intent(in) :: i
      character,intent(in) :: c
      integer :: it
      it=ichar(c)+i
      if(it < 0) it=0
      if(it > 127) it=0
      int_plus_char=char(it)
    end function int_plus_char
    function int_not(i)
      integer :: int_not
      integer,intent(in) :: i
      int_not=ieor(i,-1)
    end function
  end module
  function real_not(a)
    real :: real_not
    real,intent(in) :: a
    integer :: it
    it=transfer(a,it)
    it=ieor(it,-1)
    real_not=transfer(it,real_not)
  end function
  program interface3                    ! demonstrate usage
    use mod1
    character :: c="5"
    integer :: i=-1
    real :: r
    write(*,*) c+i                      ! calls char_plus_int
    write(*,*) -i+c                     ! calls int_plus_char
    write(*,*) i, (.not. i)             ! calls int_not
    write(*,*) tiny(-r),(.not. huge(r)) ! calls real_not
  end program
```

Example 4

```
! assignment interface
module mod4
  interface assignment (=)
    module procedure int_equals_char ! implicit interface
  end interface assignment (=)
contains
  subroutine int_equals_char(i,c) ! must have two arguments
    integer,intent(out) :: i  ! must be intent(out)
                              ! or intent(in out)
    character,intent(in) :: c ! must be intent(in)
    i=ichar(c)
  end subroutine
end module
program interface4
  use mod4
  integer :: i
  character :: c="a"
  i=c                          ! calls int_equals_char
  write(*,*) i,ichar(c)
end program
```

INTRINSIC Statement

Description

The INTRINSIC statement permits a reference to a specific intrinsic function as an actual argument.

Syntax

INTRINSIC *[::] intrinsic-procedure-names*

Where:

intrinsic-procedure-names 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 must either appear in an INTRINSIC statement or be given the intrinsic attribute in a type declaration statement in the scoping unit.

Only one appearance of a name in all of the INTRINSIC statements in a scoping unit is permitted.

A name must not appear in both an EXTERNAL and an INTRINSIC statement in the same scoping unit.

Example

```
program intrinsic
  real :: a=10.
  real,intrinsic :: log,log10 ! may be actual argument
  write(*,*) zee(a,log),zee(a,log10) ! writes 2.302585 1.0
end program
function zee(a,func)
```

```
  real :: zee,a,func
  zee=func(a)
end function
```

INVALOP Subroutine (Windows only)

Description

The INVALOP subroutine masks and detects invalid operation exceptions.

Syntax

INVALOP (*lflag*)

Arguments

lflag is an INTENT(IN OUT) scalar of type LOGICAL.

It must be set to true on the first invocation of INVALOP.

On subsequent invocations, it indicates whether an invalid operation has occurred.

Remarks

The initial invocation of the INVALOP subroutine masks the invalid operator interrupt on the floating-point unit.

Subsequent invocations return an *lflag* value of true if the exception has occurred or false if the exception has not occurred.

Example

```
logical :: lflag=.true.
call invalop(lflag) ! mask the divide-by-zero interrupt
write(*,*) lflag    ! writes F
write(*,*) 0./0.    ! writes -NaN
call invalop(lflag)
write(*,*) lflag    ! writes T
```

IOR Function

Description

The IOR function performs a bit-wise logical inclusive OR operation on two INTEGER arguments.

Syntax

IOR (*i*, *j*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

j is an INTENT(IN) scalar or array of type INTEGER and is the same kind as *i*.

Result

The result is of type INTEGER. Its value is obtained by performing a bit-wise logical inclusive OR operation on *i* and *j*.

Example

```
i=53        ! i=00110101 binary (lowest order byte)
j=45        ! j=00101101 binary (lowest order byte)
k=ior(i,j) ! k=00111101 binary (lowest order byte)
            ! k=61 decimal
```

IOSTAT_MSG Subroutine

Description

The IOSTAT_MSG subroutine retrieves text associated with a runtime error.

Syntax

IOSTAT_MSG (*iostat*, *message*)

Arguments

iostat is an INTENT(IN) scalar of type INTEGER. It contains the error status code obtained by execution of any intrinsic statement which returns a status variable.

message is an INTENT(OUT) scalar be of type CHARACTER. It is assigned the text of the runtime error message corresponding to the error code in *iostat*.

Remarks

A CHARACTER length of 256 is sufficiently large to contain all runtime error messages at this time.

If a status variable from a successful operation is passed to IOSTAT_MSG, a blank string is returned.

Example

```
real,allocatable :: a(:)
integer :: istat
```

```
character(len=256) :: msg
open(10,file="foo.bar",status="OLD",iostat=istat)
call iostat_msg(istat,msg)
write(*,*) trim(msg)
deallocate(a,stat=istat)
call iostat_msg(istat,msg)
write(*,*) trim(msg)
write(*,*) " Bye"
```

ISHFT Function

Description

The ISHFT function performs an end-off bit shift on an integer argument.

Syntax

ISHFT (*i*, *shift*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

shift is an INTENT(IN) scalar or array of type INTEGER. Its absolute value must 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 off are lost.

Example

```
integer :: i=16,ia(2)=(/4,8/)
write(*,*) i, ia       ! writes 16  4  8
write(*,*) ishft(i,-2) ! writes 4
write(*,*) ishft(i,ia) ! writes 256 4096
write(*,*) ishft(ia,2) ! writes 16   32
write(*,*) ishft(ia,i) ! writes 262144  524288
```

ISHFTC Function

Description

The ISHFTC function performs a circular shift of the rightmost bits of an integer argument.

Syntax

ISHFTC (*i*, *shift [*, *size]*)

Required Arguments

i is an INTENT(IN) scalar or array of type INTEGER, containing values to be shifted.

shift is an INTENT(IN) scalar or array of type INTEGER. The absolute value of *shift* must be less than or equal to *size*.

Optional Arguments

size is an INTENT(IN) scalar or array of type INTEGER. Only the rightmost *size* bits will be shifted.

The value of *size* must be positive and must 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 by *shift* positions.

If *shift* is positive, bits are shifted to the left.

If *shift* is negative, bits are shifted to the right.

Example

```
integer :: i=16,ia(2)=(/4,8/)
write(*,*) i,ia            ! writes 16  4  8
write(*,*) ishftc(i,-2)    ! writes 4
write(*,*) ishftc(i,ia,8)  ! writes 1 16
write(*,*) ishftc(ia,2)    ! writes 16 32
write(*,*) ishftc(ia,i,16) ! writes 4  8
```

KIND Function

Description

The KIND function returns the kind type parameter for arguments of any intrinsic type.

Syntax

KIND (*x*)

Arguments

x is an INTENT(IN) scalar or array of any intrinsic type.

If *x* has the POINTER or ALLOCATABLE attribute, it does not have to be allocated, associated, or defined.

Result

The result is a scalar INTEGER. Its value is equal to the compiler dependent kind type parameter value of *x*.

The actual values returned by the KIND function are not necessarily portable to other compiling platforms.

Example

```
! display default kinds
integer :: i
real :: r
double precision :: d
complex :: q
logical :: l
character :: c
write(*,*) "Default integer kind  : ",kind(i) ! 4
write(*,*) "Default real kind     : ",kind(r) ! 4
write(*,*) "Default dp real kind  : ",kind(d) ! 8
write(*,*) "Default complex kind  : ",kind(q) ! 4
write(*,*) "Default logical kind  : ",kind(l) ! 4
write(*,*) "Default character kind: ",kind(c) ! 1
```

LBOUND Function

Description

The LBOUND function returns the lower bounds of a whole array or a particular dimension of an array.

Syntax

LBOUND (*array [, dim]*)

Required Arguments

array is an INTENT(IN) array of any type.

It must not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Arguments
dim is an INTENT(IN) scalar of type INTEGER, with a value that is less than or equal to the rank of *array*.

Result
The result is of type default INTEGER.

If *dim* is present, the result is a scalar with 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 *array*.

The lower bound of an array section is always one. The lower bound of a zero-sized dimension is also always one.

Example
```
integer :: j(10),i(0:10,-1:10,-2:10,-3:10)
write(*,*) lbound(j)   ! writes  1
write(*,*) lbound(i)   ! writes  0 -1 -2 -3
write(*,*) lbound(i,2) ! writes -1
write(*,*) lbound(i,4) ! writes -3
```

LEN Function

Description
The LEN function returns the total length of a CHARACTER data object.

Syntax
LEN (*string*)

Arguments
string is an INTENT(IN) scalar or array of type CHARACTER.

Result
The result is a scalar default INTEGER.

Its value is the length of the character object *string*.

Example
```
character :: c1
character(len=4) :: c3="Yow!"
character(len=*), parameter :: c6=" Howdy"
character(len=3),dimension(2) :: ca2=(/" Fo","lks"/)
character :: ca4*4(2)=(/" So ","long"/)
```

```
write(*,*) len(c1),len(c3),len(c6) ! writes 1 4 6
write(*,*) len(ca2),len(ca4)       ! writes 3 4
```

LEN_TRIM Function

Description

The LEN_TRIM function returns the length of a CHARACTER string, not counting any trailing blanks.

Syntax

LEN_TRIM (*string*)

Arguments

string is an INTENT(IN) scalar or array of type CHARACTER.

Result

If *string* is scalar, the result is a scalar default INTEGER. Its value is the number of characters in *string*, not counting any trailing blanks.

If *string* is an array, the result is a conformable type default INTEGER array. Each element of the result contains the number of characters in each element of *string*, not counting any trailing blanks.

Example

```
character(len=10) :: c3="Yow!"
character(len=*),parameter :: c6="Howdy    "
character(len=3),dimension(2) :: ca2=(/"Fol","ks "/)
write(*,*) len_trim(c3),len_trim(c6) ! writes 4 5
write(*,*) len_trim(ca2)             ! writes 3 2
```

LGE Function

Description

The LGE function tests whether a string is lexically greater than or equal to another string based on the ordering of the ASCII collating sequence. See “ASCII Character Set” on page 319.

Syntax

LGE (*string_a*, *string_b*)

Arguments

string_a is an INTENT(IN) scalar or array of type CHARACTER.

string_b is an INTENT(IN) scalar or array of type CHARACTER.

If *string_a* and *string_b* are both arrays, they must have the same shape.

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; otherwise the result is false.

Trailing blanks are ignored.

If both strings are of zero length the result is true.

Example

```
character(len=3) :: a="abc",b="ABC"
character(len=0) :: a1,b1
character(len=5) :: c1(2)=(/"abc  ","123  "/)
character(len=5) :: c2(2)=(/"CBA  ","  123"/)
write(*,*) lge(a,b)   ! writes  T
write(*,*) lge(a,c1)  ! writes  T  T
write(*,*) lge(c2,a)  ! writes  F  F
write(*,*) lge(c1,c2) ! writes  T  T
```

LGT Function

Description

The LGT function tests whether a string is lexically greater than another string based on the ordering of the ASCII collating sequence. See “ASCII Character Set” on page 319.

Syntax

LGT (*string_a*, *string_b*)

Arguments

string_a is an INTENT(IN) scalar or array of type CHARACTER.

string_b is an INTENT(IN) scalar or array of type CHARACTER.

If *string_a* and *string_b* are both arrays, they must have the same shape.

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.

Trailing blanks are ignored.

If both strings are of zero length the result is false.

Example

```
character(len=3) :: a="abc",b="ABC"
character(len=0) :: a1,b1
character(len=5) :: c1(2)=(/"abc  ","123  "/)
character(len=5) :: c2(2)=(/"CBA  ","  123"/)
write(*,*) lgt(a,b)   ! writes  T
write(*,*) lgt(a,c1)  ! writes  F  T
write(*,*) lgt(c2,a)  ! writes  F  F
write(*,*) lgt(c1,c2) ! writes  T  T
```

LLE Function

Description

The LLE function tests whether a string is lexically less than or equal to another string based on the ordering of the ASCII collating sequence. See “ASCII Character Set” on page 319.

Syntax

LLE (*string_a*, *string_b*)

Arguments

string_a is an INTENT(IN) scalar or array of type CHARACTER.

string_b is an INTENT(IN) scalar or array of type CHARACTER.

If *string_a* and *string_b* are both arrays, they must have the same shape.

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; otherwise the result is false.

Trailing blanks are ignored.

If both strings are of zero length the result is true.

Example

```
character(len=3) :: a="abc",b="ABC"
character(len=0) :: a1,b1
character(len=5) :: c1(2)=(/"abc  ","123  "/)
character(len=5) :: c2(2)=(/"CBA  ","  123"/)
```

```
write(*,*) lle(a,b)   ! writes  F
write(*,*) lle(a,c1)  ! writes  T  F
write(*,*) lle(c2,a)  ! writes  T  T
write(*,*) lle(c1,c2) ! writes  F  F
```

LLT Function

Description

The LLT function tests whether a string is lexically less than another string based on the ordering of the ASCII collating sequence. See “ASCII Character Set” on page 319.

Syntax

LLT (*string_a*, *string_b*)

Arguments

string_a is an INTENT(IN) scalar or array of type CHARACTER.

string_b is an INTENT(IN) scalar or array of type CHARACTER.

If *string_a* and *string_b* are both arrays, they must have the same shape.

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.

Trailing blanks are ignored.

If both strings are of zero length the result is false.

Example

```
character(len=3) :: a="abc",b="ABC"
character(len=0) :: a1,b1
character(len=5) :: c1(2)=(/"abc  ","123  "/)
character(len=5) :: c2(2)=(/"CBA  ","  123"/)
write(*,*) llt(a,b)   ! writes  F
write(*,*) llt(a,c1)  ! writes  F  F
write(*,*) llt(c2,a)  ! writes  T  T
write(*,*) llt(c1,c2) ! writes  F  F
```

LOG Function

Description

The LOG function returns the natural logarithm of a real or complex argument.

Syntax

LOG (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL or COMPLEX.

If *x* is REAL, it must be greater than zero.

If *x* is COMPLEX, it must not be equal to zero.

Result

The result is of the same type and kind as *x*.

If *x* is REAL, its value is equal to a REAL representation of $\log_e x$.

If *x* is COMPLEX, its value is equal to the principal value with imaginary part ω in the range $-\pi < \omega \le \pi$.

If *x* is REAL and equal to zero, a floating divide exception occurs, and unless trapped, the value -Inf (negative infinity) is returned.

If *x* is REAL and less than zero, an invalid operation exception occurs, and unless trapped, the value -NaN (not a number) is returned.

If *x* is COMPLEX with both the real and imaginary parts equal to zero, a runtime error occurs and execution is terminated.

Example

```
real :: x=1.,xa(2)=(/.5,1.5/),pi=3.141592654
real :: re,im
complex :: q=(-1.,1.)
write(*,*) log(x)  ! writes 0.0
write(*,*) log(xa) ! writes -.69314718  .40546509
write(*,*) log(q)  ! writes (.34657359, 2.3561945)
re=log((sqrt(real(q)**2+aimag(q)**2))) ! real part of log(q)
im=-atan2(real(q),aimag(q))+pi/2.      ! imag part of log(q)
write(*,*) re,im
write(*,*) log(0.)  ! writes -Inf or error occurs
write(*,*) log(-1.) ! writes -NaN or error occurs
```

LOG10 Function

Description

The LOG10 function returns the common logarithm of a real argument.

Syntax

LOG10 (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

The value of *x* must be greater than zero.

Result

The result is of the same type and kind as *x*.

Its value is a REAL representation of $\log_{10} x$.

If *x* is zero, a floating divide exception occurs, and unless trapped, the value -Inf (negative infinity) is returned.

If *x* is less than zero, an invalid operation exception occurs, and unless trapped, the value -NaN (not a number) is returned.

Example

```
real :: x=1.,xa(2)=(/.5,1.5/)
logical :: true=.true., l
write(*,*) log10(x)   ! writes 0.0
write(*,*) log10(xa)  ! writes -.303103001  .17609125
write(*,*) log10(0.)  ! writes -Inf or error occurs
write(*,*) log10(-1.) ! writes -NaN or error occurs
```

LOGICAL Function

Description

The LOGICAL function converts between different kinds of data type LOGICAL.

Syntax

LOGICAL (*l [, kind]*)

Required Arguments

l is an INTENT(IN) scalar or array of type LOGICAL.

Optional Arguments

kind is INTENT(IN) and determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time.

Result

If *kind* is present, the result is of that kind; otherwise it is of default LOGICAL kind.

The result value is true if *l* is true and false if *l* is false.

Example

```
logical(kind=1) :: l1 ! not a portable declaration
logical(kind=2) :: l2 ! not a portable declaration
logical :: l4=.false.
write(*,*) logical(l4,kind(l1)) ! writes F
write(*,*) logical(l4,kind(l2)) ! writes F
```

LOGICAL Statement

Description

The LOGICAL statement declares entities having the LOGICAL data type.

Syntax

LOGICAL *[kind-selector] [[, attribute-list] ::] entity [, entity] ...*

Where:

kind-selector is (*[*KIND=*] scalar-int-initialization-expr*)

scalar-int-initialization-expr is a scalar INTEGER expression that can be evaluated at compile time.

attribute-list is a comma-separated list from the following attributes: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), EXTERNAL, INTENT(IN) or INTENT(OUT) or INTENT(IN OUT), PUBLIC or PRIVATE, INTRINSIC, OPTIONAL, POINTER, SAVE, TARGET, VOLATILE, DLL_EXPORT or DLL_IMPORT or ML_EXTERNAL.

entity is *entity-name [(array-spec)] [=initialization-expr]*
or *function-name [(array-spec)]*

array-spec is an array specification.

initialization-expr is an expression that can be evaluated at compile time.

entity-name is the name of an entity being declared.

function-name is the name of a function being declared.

Remarks

function-name must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

The =*initialization-expr* must appear if the statement contains a PARAMETER attribute.

If =*initialization-expr* appears, a double colon must appear before the list of *entities*. Each *entity* has the SAVE attribute, unless it is in a named common block.

The =*initialization-expr* must not appear if *entity-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 pointer, an external name, an intrinsic name, or an automatic object.

If an array or function with an array result is declared with the POINTER attribute, it must have a deferred shape.

If an array is declared with the ALLOCATABLE attribute, it must have a deferred shape.

If an array or function with an array result does not have the POINTER or the ALLOCATABLE attribute, it must be specified with an explicit shape.

If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attributes must not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attributes must not be specified.

The PARAMETER attribute cannot be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

The INTENT(IN), INTENT(OUT), INTENT(IN OUT), and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* may not have the PUBLIC attribute if its type has the PRIVATE attribute.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

An *entity* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

A subprogram must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

An *entity* having the ALLOCATABLE attribute cannot be a dummy argument or a function result.

An array must not have both the ALLOCATABLE attribute and the POINTER attribute.

If an *entity* has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

An *entity* may not be given the same attribute more than once in a scoping unit.

Example

```
logical,parameter :: t=.true.,f=.false. ! must be initialized
logical :: a, b(2,4)                    ! explicit shape
logical,dimension(2,2) :: c = reshape((/t,t,f,f/),shape(c))
logical,pointer :: d(:)                 ! deferred shape
logical,allocatable :: e(:)
```

MATMUL Function

Description

The MATMUL function multiplies two matrices.

Syntax

MATMUL (*matrix_a*, *matrix_b*)

Arguments

matrix_a is an INTENT(IN) array of type INTEGER, REAL, COMPLEX, or LOGICAL. It may be either rank one or two if *matrix_b* is of rank two, and must be of rank two if *matrix_b* is rank one.

matrix_b is an INTENT(IN) array of numerical type if *matrix_a* is of numerical type, and of LOGICAL type if *matrix_a* is of LOGICAL type. It may be of rank one or two if *matrix_a* is of rank two, and must be of rank two if *matrix_a* is of rank one.

The size of the first dimension must be the same as the size of the last dimension of *matrix_a*.

Result

If the arguments are of the same numeric type and kind, the result is of that type and kind. If their kind is different, the result kind is that with higher precision.

If the arguments are of different numeric types and neither is of type COMPLEX, the result is of type REAL.

If one or both of the arguments are of type COMPLEX, then the result is COMPLEX.

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 value and shape of the result are 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*, *j*) 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 :: a1(2,3),a5(5,2),b3(3),b2(2)
complex :: c2(2)
a1=reshape((/1,2,3,4,5,6/),shape(a1))
a5=reshape((/0,1,2,3,4,5,6,7,8,9/),shape(a5))
b2=(/1,2/)
b3=(/1,2,3/)
write(*,"(2i3)") a1 ! writes  1  2
                    !         3  4
                    !         5  6
write(*,*) matmul(a1,b3) ! writes 22 28
write(*,*) matmul(b2,a1) ! writes 5 11 17
write(*,"(5i3)") a5 ! writes  0  1  2  3  4
                    !         5  6  7  8  9
write(*,"(5i3)") matmul(a5,a1) ! writes 10 13 16 19 22
                                !        20 27 34 41 48
                                !        30 41 52 63 74
c2=(/(-1.,1.),(1.,-1.)/)
write(*,*) matmul(a5,c2) ! writes (5.,-5.) five times
```

MAX Function

Description

The MAX function returns the maximum value from a list of INTEGER or REAL arguments.

Syntax

MAX (*a1*, *a2*, *a3*, ...)

Arguments

The arguments are INTENT(IN) scalars or arrays of type INTEGER or REAL. They must all be of the same type and kind.

If more than one argument is an array, all arrays must have the same shape.

Result

The result is of the same type and kind as the arguments.

If all the arguments are scalar, the result is the value of the largest argument.

If any of the arguments are arrays, the result is an array with the same shape. Each element of the result is as if the scalar MAX function was called for each corresponding element of the array argument(s).

Example

```
integer :: i6(6)=(/-14,3,0,-2,19,1/)
write(*,*) max(i6,0)            ! writes 0 3 0 0 19 1
write(*,*) max(-14,3,0,-2,19,1) ! writes 19
```

MAXEXPONENT Function

Description

The MAXEXPONENT function returns the maximum binary exponent possible for a REAL argument.

Syntax

MAXEXPONENT (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is a scalar default INTEGER. Its value is the largest permissible binary exponent in the data type of *x*.

Example

```
write(*,*) maxexponent(1.e0) ! writes 128
write(*,*) maxexponent(1.d0) ! writes 1024
write(*,*) maxexponent(1.q0) ! writes 16384
```

MAXLOC Function

Description

The MAXLOC function returns the location of the first element in *array* having the maximum value of all the elements identified by *mask*.

Syntax

MAXLOC (*array [, dim] [, mask]*)

Required Arguments

array is an INTENT(IN) array of type INTEGER or REAL.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER in the range $1 \le dim \le n$, where *n* is the rank of *array*. The corresponding actual argument must not be an optional dummy argument.

mask is an INTENT(IN) scalar or array of type LOGICAL and must be conformable with *array*.

Result

The result is of type default INTEGER.

If *dim* is present, the result is an array of rank *n*-1 where *n* is the rank of *array*. The result values are the locations containing the maximum value along dimension *dim*.

If *dim* is absent, the result is an array of rank one whose size is the rank of *array*. Each element contains the subscript value 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.

Example

```
integer :: i6(6) = (/-14,3,0,-2,19,1/)
integer :: i23(2,3) = reshape((/-14,3,0,-2,19,1/),shape(i23))
write(*,'(2i4)') i23         ! writes -14   3
                             !          0  -2
                             !         19   1
write(*,*) maxloc(i6)        ! writes 5
write(*,*) maxloc(i23)       ! writes 1  3
write(*,*) maxloc(i23,dim=1) ! writes 2  1  1
write(*,*) maxloc(i23,dim=2) ! writes 3  1
write(*,*) maxloc(i23,dim=1,mask=(i23 < 10))
                             ! writes 2  1  2
```

MAXVAL Function

Description

The MAXVAL function returns the maximum value of elements of an array, along a given dimension, for which a mask is true.

Syntax

MAXVAL (*array [, dim] [, mask]*)

Required Arguments

array is an INTENT(IN) array of type INTEGER or REAL.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER in the range $1 \le dim \le n$, where *n* is the rank of *array*. The actual argument to MAXVAL must not be an optional dummy argument.

mask is an INTENT(IN) scalar or array of type LOGICAL, and must be conformable with *array*.

Result

The result is the same type and kind as *array*.

If *dim* is present, the result is an array of rank *n*-1 and of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of *array*. The results are the maximum values of all elements of *array* along dimension *dim*.

If *dim* is absent, or *array* is rank one, the result is a scalar with the value of the largest element of *array*.

If *mask* is present, the elements of *array* for which *mask* is false are not considered.

Example

```
integer :: i6(6) = (/-14,3,0,-2,19,1/)
integer :: i23(2,3) = reshape((/-14,3,0,-2,19,1/), shape(i23))
write(*,'(2i4)') i23         ! writes -14   3
                             !          0  -2
                             !         19   1
write(*,*) maxval(i6)        ! writes 19
write(*,*) maxval(i23)       ! writes 19
write(*,*) maxval(i23,dim=1) ! writes 3  0  19
write(*,*) maxval(i23,dim=2) ! writes 19 3
write(*,*) maxval(i23,dim=1,mask=(i23 < 10))
                             ! writes 3  0  1
```

MERGE Function

Description

The MERGE function chooses alternative values based on the value of a mask.

Syntax

MERGE (*tsource*, *fsource*, *mask*)

Arguments

tsource is an INTENT(IN) scalar or array and can be of any type.

fsource is an INTENT(IN) scalar or array of the same type and type parameters as *tsource*.

mask is an INTENT(IN) scalar or array of type LOGICAL.

If more than one argument is an array, all arrays must have the same shape.

Result

The result is of the same type and type parameters as *tsource*.

If all arguments are scalar, the value is *tsource* if *mask* is true, and *fsource* otherwise.

If any argument is an array, the result is an array with the same shape. Each element of the result is as if the scalar MERGE function was called for each corresponding element of the array arguments.

Example

```
integer :: i=1, j= 2
integer :: m(2,2)=reshape((/1,2,3,4/),shape(m))
integer :: n(2,2)=reshape((/4,3,2,1/),shape(n))
write(*,10) m                ! writes 1  2
                             !        3  4
write(*,10) n                ! writes 4  3
                             !        2  1
write(*,10) merge(m,n,m < n) ! writes 1  2
                             !        2  1
write(*,'(2l3)') merge(.true.,.false.,m < n) ! writes T  T
                                             !        F  F
10 format(2i3)
```

MIN Function

Description

The MIN function returns the minimum value from a list of INTEGER or REAL arguments.

Syntax

MIN (*a1*, *a2*, *a3*, ...)

Arguments

The arguments are INTENT(IN) scalars or arrays of type INTEGER or REAL. They must all be of the same type and kind.

If more than one argument is an array, all arrays must have the same shape.

Result

The result is of the same type and kind as the arguments.

If all the arguments are scalar, the result is the value of the smallest argument.

If any of the arguments are arrays, the result is an array with the same shape. Each element of the result is as if the scalar MIN function was called for each corresponding element of the array argument(s).

Example

```
integer :: i6(6)=(/-14,3,0,-2,19,1/)
write(*,*) min(i6,0)            ! writes -14 0 0 -2 0 0
write(*,*) min(-14,3,0,-2,19,1) ! writes -14
```

MINEXPONENT Function

Description

The MINEXPONENT function returns the minimum binary exponent possible for a REAL argument.

Syntax

MINEXPONENT (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is a scalar default INTEGER. Its value is the most negative binary exponent possible in the data type of *x*.

Example

```
write(*,*) minexponent(1.e0) ! writes -125
write(*,*) minexponent(1.d0) ! writes -1021
write(*,*) minexponent(1.q0) ! writes -16381
```

MINLOC Function

Description

The MINLOC function returns the location of the first element in *array* having the minimum value of all the elements identified by *mask*.

Syntax

MINLOC (*array*, *dim*, *mask*)

Required Arguments

array is an INTENT(IN) array of type INTEGER or REAL.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER in the range $1 \leq dim \leq n$, where *n* is the rank of *array*. The corresponding actual argument must not be an optional dummy argument.

mask is an INTENT(IN) scalar or array of type LOGICAL, and must be conformable with *array*.

Result

The result is of type default INTEGER.

If *dim* is present, 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.

Example

```
integer :: i6(6)=(/-14,3,0,-2,19,1/)
integer :: i23(2,3)=reshape((/-14,3,0,-2,19,1/),shape(i23))
write(*,'(2i4)') i23                ! writes -14  3
                                    !          0 -2
                                    !         19  1
write(*,*) minloc(i6)               ! writes 1
write(*,*) minloc(i23)              ! writes 1  1
write(*,*) minloc(i23,1)            ! writes 1  2  2
write(*,*) minloc(i23,2)            ! writes 1  2
write(*,*) minloc(i23,1,(i23 < 10)) ! writes 1  2  2
```

MINVAL Function

Description

The MAXVAL function returns the minimum value of elements of an array, along a given dimension, for which a mask is true.

Syntax

MINVAL (*array [, dim] [, mask]*)

Required Arguments

array is an INTENT(IN) array of type INTEGER or REAL.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER in the range $1 \leq dim \leq n$, where *n* is the rank of *array*. The actual argument to MINVAL must not be an optional dummy argument.

mask is an INTENT(IN) scalar or array of type LOGICAL, and must be conformable with *array*.

Result

The result is the same type and kind as *array*.

If *dim* is present, the result is an array of rank *n*-1 and of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of *array*. The results are the minimum values of all elements of *array* along dimension *dim*.

If *dim* is absent, or *array* is rank one, the result is a scalar with the value of the smallest element of *array*.

If *mask* is present, the elements of *array* for which *mask* is false are not considered.

Example

```
integer :: i6(6)=(/-14,3,0,-2,19,1/)
integer :: i23(2,3)=reshape((/-14,3,0,-2,19,1/),shape(i23))
write(*,'(2i4)') i23                ! writes -14  3
                                    !          0 -2
                                    !         19  1
write(*,*) minloc(i6)               ! writes   1
write(*,*) minloc(i23)              ! writes   1  1
write(*,*) minloc(i23,1)            ! writes   1  2  2
write(*,*) minloc(i23,2)            ! writes   1  2
write(*,*) minloc(i23,1,(i23 < 10)) ! writes   1  2  2
```

ML_EXTERNAL Statement

Description

The ML_EXTERNAL statement makes a procedure externally available to a statically linked mixed-language calling routine.

Syntax

ML_EXTERNAL *[::] ml-external-names*

Where:

ml-external-names is a list of procedures defined in the current scoping unit.

Remarks

The procedures in *ml-external-names* must not be module procedures.

The procedures names listed in an ML_EXTERNAL statement are "decorated" to match one of several calling conventions by using the "-ml xxxx" switch at compile time.

Example

```
function half(x)
  integer :: half,x
  ml_external half              ! ml_external statement
  half=x/2
end function half
function twice(x)
  integer,ml_external :: twice ! ml_external attribute
  integer :: x
  twice=x*2
end function twice
```

MOD Function

Description

The MOD function returns the remainder from the division of the first argument by the second argument.

Syntax

MOD (*a*, *p*)

Arguments

a is an INTENT(IN) scalar or array of type INTEGER or REAL.

p is INTENT(IN) and of the same type and kind as *a*. Its value must not be zero.

Result

The result is the same type and kind as *a*. Its value is *a* - INT(*a* / *p*) * *p*.

Example

```
write(*,*) mod(23.4,4.0) ! writes  3.4
write(*,*) mod(-23,4)    ! writes  -3
write(*,*) mod(23,-4)    ! writes  3
write(*,*) mod(-23,-4)   ! writes  -3
```

MODULE Statement

Description

The MODULE statement begins a module program unit. The module encapsulates data and procedures, provides a global data facility, which can be considered a replacement for COMMON, and establishes implicit interfaces for procedures contained in the module.

Syntax

MODULE *module-name*

Where:

module-name is the name of the module.

Remarks

The module name must not be the same as the name of another program unit, an external procedure, or a common block within the executable program, nor be the same as any local name in the module.

In LF95, a module program unit must be compiled before it is used.

Example

```
module m
  type mytype         ! mytype available anywhere m is used
    real :: a,b(2,4)
    integer :: n,o,p
  end type mytype
  real :: r1=1      ! r1 available anywhere m is used
contains
  subroutine sub1(i)  ! implicit interface for sub1
    integer :: i
    i=1
  end subroutine
  function fun1()     ! implicit interface for fun1
    integer :: fun1
    fun1=1
  end function
end module m
program zee
  use m              ! makes module available to program zee
  type (mytype) bee,dee
  integer :: i
  i=fun1()
  call sub1(i)
end program zee
```

MODULE PROCEDURE Statement

Description

The MODULE PROCEDURE statement specifies that the names in the *module-procedure-list* are part of a generic interface.

Syntax

MODULE PROCEDURE *module-procedure-list*

Where:

module-procedure-list is a list of module procedures accessible by host or use association.

Remarks

A MODULE PROCEDURE statement can only appear in a generic interface block within a module or within a program unit that accesses a module by use association.

Example

```
module mod1
  interface swap
    module procedure complex_swap ! interface for a module
    module procedure logical_swap ! procedure is implicit
  end interface
contains
  subroutine complex_swap(cx,cy) ! interface is defined here
    complex :: cx,cy,ct
    ct=cx
    cx=cy
    cy=ct
  end subroutine
  subroutine logical_swap(lx,ly) ! interface is defined here
    logical :: lx,ly,lt
    lt=lx
    lx=ly
    ly=lt
  end subroutine
end module
```

MODULO Function

Description

The MODULO function returns the modulo of two numbers.

Syntax

MODULO (*a*, *p*)

Arguments

a is an INTENT(IN) scalar or array of type INTEGER or REAL.

p is INTENT(IN) and must be of the same type and kind as *a*. Its value must 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 chosen so that *r* is nearer to zero than *p*.

Example

```
r=modulo(23.4,4.0) ! r is assigned the value 3.4
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
```

MVBITS Subroutine

Description

The MVBITS subroutine copies a sequence of bits from one INTEGER data object to another.

Syntax

MVBITS (*from*, *frompos*, *len*, *to*, *topos*)

Arguments

from is an INTENT(IN) scalar or array of type INTEGER.

frompos is an INTENT(IN) scalar or array of type INTEGER. It must be non-negative. *frompos* + *len* must be less than or equal to BIT_SIZE(*from*).

len is an INTENT(IN) scalar or array of type INTEGER. It must be non-negative.

to is an INTENT(IN OUT) scalar or array of type INTEGER with the same kind as *from*. It can be the same variable as *from*.

topos is an INTENT(IN) scalar or array of type INTEGER and must be non-negative. *topos* + *len* must be less than or equal to BIT_SIZE(*to*).

Remarks

to is set by copying *len* bits, starting at position *frompos*, from *from*, to *to*, starting at position *topos*.

If any of *from*, *frompos*, *len* or *topos* are arrays, *to* must be an array with the same shape.

If *to* is an array, its value is as if the scalar MVBITS operation were performed on each corresponding element of any array arguments.

Example

```
integer :: i; data i/z'0f0f'/
integer :: ia(2)=(/2,4/),ja(2)
write(*,"(b32)") i      ! writes 111100001111
call mvbits(i,0,4,i,4)
write(*,"(b32)") i      ! writes 111111111111
call mvbits(i,ia,4,ja,ia)
write(*,"(b32)") ja     ! writes       111100
                        ! writes     11110000
```

NAMELIST Statement

Description

The NAMELIST statement specifies a list of variables that can be referred to by one name for the purpose of performing input/output.

Syntax

NAMELIST /*name*/ *group [[,]* /*name*/ *group] ...*

Where:

name is the name of a namelist group.

group is a list of variable names.

Remarks

A name in a *group* must not be the name of an array dummy argument with a non-constant bound, a variable with a non-constant character length, an automatic object, a pointer, a variable of a type that has an ultimate component that is a pointer, or an allocatable array.

If a *name* has the public attribute, no item in *group* can have the PRIVATE attribute.

The order in which the variables appear in a NAMELIST statement determines the order in which the variables' values will appear on output.

Example

```
real :: a,b
integer :: i,j
namelist /input/ a,b,i,j
open(10,file='data.dat')
read(10,nml=input)
write(*,nml=input)
close(10)
end

! namelist data file
&input
b=12.3
i=4
a=13.2
j=12
/
```

NDPERR Function (Windows Only)

Description

The NDPERR function detects exceptions raised by the numeric data processor.

Syntax

NDPERR (*lvar*)

Arguments

lvar is an INTENT(IN) scalar of type LOGICAL.

If *lvar* is true, NDPERR clears floating-point exception bits.

If *lvar* is false, NDPERR does not clear floating-point exception bits.

Result

The result is of type default INTEGER. Its value is the INTEGER value of the combination of the following bits, where a bit set to one indicates an exception has occurred:

Table 10: NDPERR bits

Bit	Exception
0	Invalid Operation
1	Denormalized Number
2	Divide by Zero
3	Overflow
4	Underflow

Example

```
write(*,*) 0./0.                ! writes -NaN
write(*,*) ndperr(.true.)       ! writes 1
write(*,*) tiny(1.e0)/100000.  ! writes 1.1770907e-43
write(*,*) ndperr(.true.)       ! writes 2
```

NDPEXC Subroutine (Windows Only)

Description

The NDPEXC subroutine masks exceptions raised by the numeric data processor.

Arguments

The NDPEXC subroutine has no arguments.

Remarks

To mask specific exceptions use the subroutines INVALOP (invalid operator), OVEFL (overflow), UNDFL (underflow), and DVCHK (divide by zero).

The precision exception is always masked.

Example

```
call ndpexc ()  ! mask floating-point exceptions
```

NEAREST Function

Description

The NEAREST function returns the nearest number of a given data type in a given direction.

Syntax

NEAREST (*x*, *s*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

s is an INTENT(IN) scalar or array of type REAL. It must be non-zero.

Result

The result is REAL and of the same kind as *x*.

If both *x* and *s* are scalar, the result value is the distinct number nearest to *x*, in the direction indicated by the sign of *s*.

Example

```
real(kind(1.e0)) :: r1=1.e0
real(kind(1.d0)) :: r2=1.d0
write(*,*) r1              ! writes 1.00000000
write(*,*) nearest(r1,1.)  ! writes 1.00000012
write(*,*) nearest(r1,-1.) ! writes 0.99999994
write(*,"(3z10.8)") r1              ! writes 3f800000
write(*,"(3z10.8)") nearest(r1,1.)  ! writes 3f800001
write(*,"(3z10.8)") nearest(r1,-1.) ! writes 3f7fffff
write(*,*) r2              ! writes 1.00000000000000000
write(*,*) nearest(r2,1.)  ! writes 1.00000000000000000
write(*,*) nearest(r2,-1.) ! writes 0.99999999999999999
write(*,"(z18.16)") r2             ! writes 3ff0000000000000
write(*,"(z18.16)") nearest(r2,1.) ! writes 3ff0000000000001
write(*,"(z18.16)") nearest(r2,-1.)! writes 3f7fffffffffffff
```

NINT Function

Description

The NINT function returns the nearest INTEGER to a REAL argument.

Syntax

NINT (*a [, kind]*)

Required Arguments

a is an INTENT(IN) scalar or array of type REAL.

Optional Arguments

kind is INTENT(IN) and determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *“KIND Function”* or *“SELECTED_INT_KIND Function”*.

Result

The result is of type INTEGER. If *kind* is present the result is that kind; otherwise it is a default INTEGER.

If $a > 0$, the result has the value INT(a + 0.5);

If $a \leq 0$, the result has the value INT(a - 0.5).

Example

```
real :: a=1.5,aa(3)=(/-.5,0.,.5/)
write(*,*) nint(a)   ! writes 2
write(*,*) nint(-a)  ! writes -2
write(*,*) nint(aa)  ! writes -1 0 1
write(*,*) nint(-aa) ! writes 1 0 -1
```

NOT Function

Description

The NOT function returns the bit-wise logical complement of an INTEGER argument.

Syntax

NOT (*i*)

Arguments

i is an INTENT(IN) scalar or array of type INTEGER.

Result

The result is an INTEGER of the same 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

```
integer :: ia(3)=(/-1,0,1/)
write(*,*) not(-1) ! writes 0
write(*,*) not(0)  ! writes -1
write(*,*) not(ia) ! writes 0 -1 -2
```

NULL Function

Description

The NULL function returns a disassociated pointer.

Syntax

NULL (*[mold]*)

Optional Argument

mold must be a pointer and may be of any type.

mold must be present when a reference to NULL() appears as an actual argument in a reference to a generic procedure if the type, type parameters, or rank is required to resolve the generic reference.

Result

A disassociated pointer of the same type, type parameters, and rank as the pointer that becomes associated with the result.

Example

```
real,pointer,dimension(:) :: a => null() ! a is disassociated
```

NULLIFY Statement

Description

The NULLIFY statement disassociates a pointer.

Syntax

NULLIFY (*pointers)*

Where:

pointers is a comma-separated list of variables or structure components having 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
```

OPEN Statement

Description

The OPEN statement connects or reconnects an external file to an input/output unit.

Syntax

OPEN (*connect-specs*)

Where:

connect-specs is a comma-separated list of
[UNIT =] external-file-unit
or IOSTAT=*iostat*
or 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 BLOCKSIZE=*blocksize*
or CONVERT =*file-format*
or CARRIAGECONTROL=*carriagecontrol*

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.

iostat is a scalar default INTEGER variable that is assigned a positive value if an error condition occurs, a negative value if an end-of-file or end-of-record condition occurs, and zero otherwise.

label is the statement label of the statement that is branched to if an error occurs.

status is a scalar default CHARACTER expression. It must 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, if the file is to be opened but not replaced; and UNKNOWN otherwise. The default is UNKNOWN.

access is a scalar default CHARACTER expression. It must evaluate to SEQUENTIAL if the file is to be connected for sequential access, DIRECT if the file is to be connected for direct access, or TRANSPARENT if the file is to be connected for binary (transparent) access. The default value is SEQUENTIAL

form is a scalar default CHARACTER expression. It must 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, or BINARY if the file is to be connected for binary (transparent) access. 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 must 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.

blank is a scalar default CHARACTER expression. It must evaluate to NULL if null blank control is used and ZERO if zero blank control is used. The default value is NULL. This specifier is only permitted for a file being connected for formatted input/output.

position is a scalar default CHARACTER expression. It must 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. Note that the POSITION keyword may only be used for sequential access files.

action is a scalar default CHARACTER expression. It must 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 if the file is to be connected for input and output. The default value is READWRITE.

delim is a scalar default CHARACTER expression. It must evaluate to APOSTROPHE if the apostrophe is used to delimit character constants written with list-directed or namelist formatting, QUOTE if the quotation mark is used, and NONE if neither quotation marks nor apostrophes is used. The default value is NONE. This specifier is permitted only for formatted files and is ignored on input.

pad is a scalar default CHARACTER expression. It must evaluate to YES if the formatted input record is to be padded with blanks and NO otherwise. The default value is YES.

blocksize is a scalar default INTEGER expression. It must evaluate to the size, in bytes, of the input/output buffer.

file-format is a scalar default CHARACTER variable that evaluates to BIG_ENDIAN if big endian conversion is to occur, LITTLE_ENDIAN if little endian conversion is to occur, IBM if IBM style conversion is to occur, and NATIVE if no conversion is to occur.

carriagecontrol is a scalar default CHARACTER expression. It must evaluate to FORTRAN if the first character of a formatted sequential record used for carriage control, and LIST otherwise. Non-storage devices default to FORTRAN; disk files to LIST

Remarks

The OPEN statement connects an existing file to an input/output unit, creates a file that is preconnected, creates a file and connects it to an input/output unit, or changes certain characteristics of a connection between a file and an input/output unit.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the connect-spec-list.

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 connected to an input/output unit, it may not be opened with a different unit number.

FILE= is optional if it is the second argument and the first argument is a unit number with no UNIT=.

A unit opened for BINARY or TRANSPARENT access is open for direct access with a record length of one, so REC= may appear, and END= may not appear in any i/o statement for the unit.

Example

```
integer :: ierr
open(8,"inf.dat",status="new") ! creates a new file
                               ! error if file exists
open(9,file="open.f90",status="old") ! file must exist
open(10,status="scratch")      ! file deleted on close
open(11,"inf.dat",iostat=ierr) ! ierr returns status
                               ! same file cannot be
                               ! open on two units
open(file="foo",      & ! if unit is not first, must
     unit=12,         & ! have "unit=" keyword
     access="direct", & ! direct access requires recl
     recl=10)
```

OPTIONAL Statement

Description

The OPTIONAL statement declares that any dummy arguments specified need not be associated with an actual argument when the procedure is invoked.

Syntax

OPTIONAL *[::] dummy-arg-names*

Where:

dummy-arg-names is a comma-separated list of dummy argument names.

Remarks

A OPTIONAL dummy argument is tested for presence by using the *“PRESENT Function”*.

An optional dummy argument must not be referenced unless it has been tested and found to be present.

An interface is required before any procedure that has optional arguments can be called.

An optional argument may not be an actual argument if the corresponding dummy argument is not optional.

Example

```
module mod1             ! provides implicit interface
contains
  function fun1(a,b)
    real :: fun1,a
    real,optional :: b  ! optional attribute
    if(present(b)) then ! don’t reference b unless
      fun1=a+b          ! it is present
    else
      fun1=a
    end if
  end function
end module
program present
  use mod1
  write(*,*) fun1(2.)    ! no optional argument
  write(*,*) fun1(2.,2.) ! optional argument
end program
```

OVEFL Subroutine (Windows Only)

Description

The OVEFL subroutine masks and detects floating-point overflow exceptions.

Syntax

OVEFL (*lflag*)

Arguments

lflag must be of type LOGICAL. It is assigned the value true if an overflow exception has occurred, and false otherwise.

Remarks

lflag must be set to true on the first invocation.

The initial invocation of the OVEFL subroutine masks the overflow interrupt on the floating-point unit.

Subsequent invocation returns an *lflag* value of true if the exception has occurred or false if the exception has not occurred.

Example

```
real(kind(1.d0)) :: a=huge(a)
logical :: lflag = .true.
call ovefl(lflag)  ! mask the overflow interrupt
write(*,*) lflag   ! writes F
do
  a=a*2.d0
  call ovefl(lflag)! test for overflow
  if(lflag) exit
end do
write(*,*) lflag   ! writes T
```

PACK Function

Description

The PACK function packs an array into a vector under the control of a mask.

Syntax

PACK (*array*, *mask [*, *vector]*)

Required Arguments

array is an INTENT(IN) array can be of any type.

mask is INTENT(IN) and must be of type LOGICAL. *mask* must be conformable with *array*.

Optional Arguments

vector is an INTENT(IN) array of rank one, and must be the same type and kind as *array*. It must have at least as many elements as there are true elements in *array*. If *mask* is scalar with value true, *vector* must 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 *i*th 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*.

Example

```
integer :: c(3,3)=reshape((/0,3,2,4,3,2,5,1,2/),shape(c))
integer :: cc(9)=-1
write(*,'(3i3)') c        ! writes 0 3 2
                          !        4 3 2
                          !        5 1 2
write(*,*) pack(c,mask=(c > 2))
                          ! writes 3 4 3 5
write(*,*) pack(c,mask=(c > 2),vector=cc)
                          ! writes 3 4 3 5 -1 -1 -1 -1 -1
write(*,*) pack(c,.true.) ! writes 0 3 2 4 3 2 5 1 2
```

PARAMETER Statement

Description

The PARAMETER statement specifies and initializes named constants.

Syntax

PARAMETER (*named-constant-defs*)

Where:

named-constant-defs is a comma separated list of *constant-name=init-expr*

constant-name is the name of a constant being specified.

init-expr is an expression that can be evaluated at compile time.

Remarks

Each named constant becomes defined with the value of *init-expr*.

Any data objects defined in a PARAMETER statement cannot be subsequently redefined.

Example

```
real, parameter :: pi=3.141592654 ! parameter attribute
integer :: i0
parameter (i0=0)                  ! parameter statement
```

PAUSE Statement (obsolescent)

Description

The PAUSE statement temporarily suspends execution of the program.

Syntax

PAUSE

Remarks

When a PAUSE statement is reached, the string "`Press any key to continue`" is displayed. The program resumes execution when a key representing any printable character is pressed.

The PAUSE statement was considered obsolescent in Fortran 90, and has been deleted from the Fortran 95 language specification. Regardless of this, the PAUSE statement will continue to be supported by LF95.

Example

```
pause     !"Press any key to continue . . ." is displayed
```

Pointer Assignment Statement

Description

The pointer assignment statement associates a pointer with a target.

Syntax

pointer => *target*

Where:

pointer is a variable having the POINTER attribute.

target is a variable or expression having the TARGET or POINTER attribute, or is a subobject of a variable having the TARGET attribute.

Remarks

If *target* is not a pointer, *pointer* becomes associated with *target*.

If *target* is an associated pointer, *pointer* becomes associated with the same object as *target*.

If *target* is disassociated, *pointer* becomes disassociated.

If *target*'s association status is undefined, *pointer*'s also becomes undefined.

Pointer assignment of a pointer component of a structure can also take place by derived type intrinsic assignment or by a defined assignment.

When a pointer assignment statement is executed, any previous association of *pointer* is broken.

target must be of the same type, kind, and rank as *pointer*.

target must not be an array section with a vector subscript.

If *target* is an expression, it must deliver a pointer result.

Example

```
real,pointer :: a => null(),b => null()
real,target :: c=5.0
a => c  ! a is an alias for b
b => a  ! b is an alias for a (and c)
write(*,*) a,b,c
```

POINTER Function

Description

The POINTER function gets the memory address of a variable, substring, array reference, or external subprogram.

Syntax

POINTER (*item*)

Arguments

item can be of any type. It is the name for which to return an address. *item* must have the EXTERNAL attribute.

Result

The result is of type INTEGER. It is the address of *item*.

Example

```
real :: a,b(10)
write(*,*) pointer(a)    ! writes the memory
write(*,*) pointer(b)    ! address of each of
write(*,*) pointer(b(2)) ! these variables
end
```

POINTER Statement

Description

The POINTER statement specifies a list of variables that have the POINTER attribute.

Syntax

POINTER *[::] variable-name [*(*deferred-shape*)*] [, variable-name [*(*deferred-shape*)*]] ...*

or (Cray pointer)

POINTER (*int-var, target-var*) *[,* (*int-var, target-var*) *...]*

Where:

variable-name is the name of a variable.

deferred-shape is : *[, :] ...* where the number of colons is equal to the rank of *variable-name.*

int-var is assumed to be an INTEGER variable, and cannot appear in a type declaration statement.

target-var is the target variable that *int-var* will be an alias for.

Remarks

A pointer must not be referenced or defined unless it is first associated with a target through a pointer assignment or an ALLOCATE statement.

The INTENT attribute must not be specified for a variable having the POINTER attribute.

If the DIMENSION attribute is specified elsewhere in the scoping unit, the array must have a deferred shape.

int-var cannot also appear as a *target-var*.

int-var and *target-var* cannot also have the ALLOCATABLE, INTRINSIC, EXTERNAL, PARAMETER, POINTER or TARGET attributes.

Cray pointers are provided for compatibility purposes, and should not be used when writing new code.

Example

```
integer,pointer :: index(:)   ! pointer attribute
real :: next,previous,r1(20)
pointer :: next(:,:),previous ! pointer statement
pointer (i,j),(k,r1)          ! Cray pointers
```

PRECFILL Subroutine

Description

Set fill character for numeric fields that are wider than supplied numeric precision. The default is ’0’.

Syntax

PRECFILL (*filchar*)

Arguments

filchar is INTENT(IN) and of type CHARACTER. The first character becomes the new precision fill character.

Example

```
call precfill('*')  ! '*' is the new precision fill character
```

PRECISION Function

Description

The PRECISION function returns the decimal precision of a REAL or COMPLEX data type.

Syntax

PRECISION (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL or COMPLEX.

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*.

Example

```
real(kind(1.e0)) :: r10
real(kind(1.d0)) :: r100
real(kind(1.q0)) :: r1000
write(*,*) precision(r10)   ! writes 6
write(*,*) precision(r100)  ! writes 15
write(*,*) precision(r1000) ! writes 33
```

PRESENT Function

Description

The PRESENT function determines whether or not an optional argument is present.

Syntax

PRESENT (*a*)

Arguments

a is INTENT(IN) and must be an optional dummy argument of the procedure in which the PRESENT function appears.

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; otherwise, it is false.

Example

```
function fun1(a,b)
  real :: fun1,a
  real,optional :: b
  if(present(b)) then ! don’t reference b unless
    fun1=a+b          ! it is present
  else
    fun1=a
  end if
end function
```

PRINT Statement

Description

The PRINT statement writes values from an output list to the console.

Syntax

PRINT *format [, outputs]*

Where:

format is *format-expr*
or *label*
or *
or *assigned-label*

format-expr is a default CHARACTER expression that evaluates to (*[format-items]*)

label is a statement label of a FORMAT statement.

assigned-label is a scalar default INTEGER variable that was assigned the label of a FORMAT statement in the same scoping unit.

outputs is a comma-separated list of *expr*
or *io-implied-do*

expr is an expression.

io-implied-do is (*outputs, implied-do-control*)

implied-do-control is *do-variable=start, end [, increment]*

start, *end*, and *increment* are scalar numeric expressions of type INTEGER, REAL or double-precision REAL.

do-variable is a scalar variable of type INTEGER, REAL or double-precision REAL.

format-items is a comma-separated list of *[r]data-edit-descriptor*, *control-edit-descriptor*, or *char-string-edit-descriptor*, or *[r]*(*format-items*)

data-edit-descriptor is any valid format descriptor. See “FORMAT Statement” on page 139.
char-string-edit-descriptor is a CHARACTER literal constant or *c*H*rep-chars*

rep-chars is a string of characters

c is the number of characters in *rep-chars*

r, *k*, and *n* are positive INTEGER literal constants that are used to specify a number of repetitions of the *data-edit-descriptor, char-string-edit-descriptor, control-edit-descriptor*, or (*format-items*)

Remarks

The *do-variable* of an *implied-do-control* that is contained within another *io-implied-do* must not appear as the *do-variable* of the containing *io-implied-do*.

If an array appears as an output item, it is treated as if the elements are specified in array-element order.

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.

The comma used to separate items in *format-items* can be omitted between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor; before a slash edit descriptor when the optional repeat specification is not present; after a slash edit descriptor; and before or after a colon edit descriptor.

Within a CHARACTER literal constant, if an apostrophe or quotation mark appears, it must be as a consecutive pair without any blanks. Each such pair represents a single occurrence of the delimiter character.

Example

```
    integer :: i=1,j=2,k=3
    print *," i =",i," j =",j," k =",k
    print "(3i8)",i,j,k
    print 100,i,j,k
100 format(3i8)
```

PRIVATE Statement

Description

The PRIVATE statement specifies that the names of entities are accessible only within the current module.

Syntax

PRIVATE *[[::] access-ids]*

Where:

access-ids is a comma-separated list of
use-name
or *generic-spec*

use-name is a name previously declared in the module.

generic-spec is *generic-name*
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

generic-name is the name of a generic procedure.

defined-operator is one of the intrinsic operators or *.op-name*.

op-name is a user-defined name for the operation.

Remarks

The PRIVATE statement is permitted only in a module.

If the PRIVATE statement appears without a list of objects, it sets the default accessibility of named items in the module to private; otherwise, it makes the accessibility of the listed objects private.

If the PRIVATE statement appears in a derived type definition, the entities within the derived type definition are accessible only in the current module. Within a derived type definition, the PRIVATE statement cannot have an object list.

Example

```
module ex
  implicit none
  ! default accessibility is public
  real :: a,b
  private a         ! a is not accessible outside module
                    ! b is accessible outside module
  real,private :: c ! private attribute
  type zee
    private
    integer :: l,m  ! zee, l and m are private
  end type zee
end module ex
```

PRODUCT Function

Description

The PRODUCT function returns the product of elements of an array expression, along a given dimension, or under the control of a logical mask.

Syntax

PRODUCT (*array [, dim] [, mask]*)

Required Arguments

array is an INTENT(IN) array of type INTEGER, REAL or COMPLEX.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER in the range $1 \le dim \le n$, where *n* is the rank of *array*. The corresponding actual argument must not be an optional dummy argument.

mask is INTENT(IN), must be of type LOGICAL, and must 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 $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of *array*.

If *dim* is absent, the result is the product of all the elements of *array*.

If *dim* is present, the result is the product of all elements of *array* along dimension *dim*.

If *mask* is present, the result is the product of all elements of *array* for which *mask* evaluates to true.

Example

```
integer,dimension(2,2) :: m=reshape((/1,2,3,4/),shape(m))
write(*,'2i3)') m                    ! writes 1  2
                                     !        3  4
write(*,*) product(m)                ! writes 24
write(*,*) product(m,dim=1)          ! writes 2 12
write(*,*) product(m,dim=2)          ! writes 3 8
write(*,*) product(m,mask=m>2)       ! writes 12
write(*,*) product(m,dim=1,mask=m>2) ! writes 1 12
write(*,*) product(m,dim=2,mask=m>2) ! writes 3 4
```

PROGRAM Statement

Description

The PROGRAM statement signals the beginning of a main program unit.

Syntax

PROGRAM *program-name*

Where:

program-name is the name given to the main program.

Remarks

program-name is global to the entire executable program. It must not be the same as the name of another program unit, external procedure, or common block in the executable program. It may not be the same as any local name in the main program.

Example

```
program zyx
  ! code goes here
end program zyx
```

PUBLIC Statement

Description

The PUBLIC statement specifies that entities are accessible by use association anywhere the module that contains the PUBLIC statement is used.

Syntax

PUBLIC *[[::] access-ids]*

Where:

access-ids is a comma-separated list of *use-name*
or *generic-spec*

use-name is any name within the scope of the module in which the PUBLIC statement appears.

generic-spec is *generic-name*
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

generic-name is the name of a generic procedure.

defined-operator is one of the intrinsic operators
or *.op-name*.

op-name is a user-defined name for the operation.

Remarks

The PUBLIC statement is permitted only within a module.

The default accessibility of names in a module is public. 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 objects specified accessible both within the module, and to any procedure that uses that module.

Example

```
module zee
  implicit none
  private          ! default accessibility is now private
  real :: a,b
  public a         ! a is now accessible outside module
  real,public :: c ! public attribute
end module zee
```

PURE Procedure

Description

A PURE procedure declaration ensures that no unseen side effects will occur upon invocation of the procedure.

Syntax

PURE SUBROUTINE *sub-name* (*arg-list*)

or

PURE FUNCTION *fun_name* (*arg-list*) *[*result(*result-var*)*]*

Where:

sub-name is the subroutine name

fun-name is the function name.

arg-list is a list of dummy arguments.

result-var defines the type and kind of the result, and is assigned the result value.

Remarks

If the PURE procedure is a subroutine, then each argument in *arg-list* must declare the INTENT attribute, unless that argument corresponds to a procedure, is an alternate return, or has the POINTER attribute.

If the PURE procedure is a function, then each argument in *arg-list* must be declared as INTENT(IN) unless that argument corresponds to a procedure or has the POINTER attribute.

Local variables within the scope of a PURE procedure cannot have the SAVE attribute, which implies that they cannot be initialized when declared, or by a DATA statement.

Any procedures (including dummy procedures) that are invoked from a PURE procedure must be PURE.

Local variables of pure subroutines must not have the SAVE attribute, either by explicit declaration or by initialization in a type declaration or DATA statement.

Any subprogram contained within a PURE procedure is also PURE.

A PURE procedure may not cause the value of a variable which is in COMMON, or is available by use or host association to be altered.

No external I/O operations may occur within a PURE procedure.

A PURE procedure may not contain a STOP statement.

Example

```
pure subroutine sub1(a)
  real,intent(in out) :: a ! intent must be declared
  interface
    pure function fun1(a) ! any invoked procedure must be pure
      real,intent(in) :: a
    end function fun1
  end interface
  a=fun1(a/10.)
end subroutine
pure function fun1(a)
  real :: fun1
  real,intent(in) :: a ! all arguments must be intent(in)
  fun1=a
end function fun1
```

RADIX Function

Description

The RADIX function returns the number base of the physical representation of a number.

Syntax

RADIX (*x*)

Arguments

x must be of type INTEGER or REAL.

Result

The result is a default INTEGER scalar whose value is the number base of the physical representation of *x*. This value is two for all kinds of INTEGERs and REALs.

Example

```
write(*,*) radix(2.3) ! writes 2
```

RANDOM_NUMBER Subroutine

Description

The RANDOM_NUMBER subroutine returns a uniformly distributed pseudorandom number or numbers in the range $0 \leq x < 1$.

Syntax

RANDOM_NUMBER (*harvest*)

Arguments

harvest is an INTENT(OUT) scalar or array of type REAL. On return, its value is a set of pseudorandom numbers uniformly distributed in the range $0 \leq x < 1$.

Remarks

The random number generator uses a multiplicative congruential algorithm with a period of approximately 2^{38}

Example

```
real,dimension(8) :: x
call random_number(x) ! each element of x is assigned
                      ! a pseudorandom number
```

RANDOM_SEED Subroutine

Description

The RANDOM_SEED subroutine initializes or queries the pseudorandom number generator used by RANDOM_NUMBER.

Syntax

RANDOM_SEED (*[*size=*size] [*put=*put] [*get=*get]*)

Optional Arguments

size is an INTENT(OUT) scalar of type default INTEGER. It is set to the number of default INTEGERs the processor uses to hold the seed. For LF95 this value is one.

put is an INTENT(IN) default INTEGER array of rank one and size greater than or equal to *size*. It is used by the processor to set the seed value.

get is an INTENT(OUT) default INTEGER array of rank one and size greater than or equal to *size*. It is set to the current value of the seed.

Remarks

The RANDOM_SEED subroutine can only be called with one or zero arguments.

If no argument is present, the system generates a seed value and initializes the random number generator.

Example

```
integer :: seed_size
integer,allocatable :: seed(:)
call random_seed() ! initialize with system generated seed
call random_seed(size=seed_size) ! find out size of seed
allocate(seed(seed_size))
call random_seed(get=seed) ! get system generated seed
write(*,*) seed            ! writes system generated seed
seed=314159265
call random_seed(put=seed) ! set current seed
call random_seed(get=seed) ! get current seed
write(*,*) seed            ! writes 314159265
deallocate(seed)           ! safe
```

RANGE Function

Description

The RANGE function returns the decimal range of any numeric data type.

Syntax

RANGE (*x*)

Arguments

x is an INTENT(IN) scalar or array of any numeric type.

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*)))).

Example

```
real(kind(1.e0)) :: r10
real(kind(1.d0)) :: r100
real(kind(1.q0)) :: r1000
integer(selected_int_kind(r=1)) :: i1
integer(selected_int_kind(r=4)) :: i4
integer(selected_int_kind(r=7)) :: i7
integer(selected_int_kind(r=12)) :: i12
write(*,*) range(r10)   ! writes 37
write(*,*) range(r100)  ! writes 307
write(*,*) range(r1000) ! writes 4931
write(*,*) range(i1)    ! writes 2
write(*,*) range(i4)    ! writes 4
write(*,*) range(i7)    ! writes 9
write(*,*) range(i12)   ! writes 18
```

READ Statement

Description

The READ statement transfers values from an input/output unit to the data objects specified in an input list or a namelist group.

Syntax

READ (*io-control-specs*) *[inputs]*

or

READ *format [, inputs]*

Where:

inputs is a comma-separated list of *variable*
or *io-implied-do*

variable is a variable.

io-implied-do is (*inputs*, *implied-do-control*)

implied-do-control is *do-variable*=*start*, *end [*, *increment]*

start, *end*, and *increment* are scalar numeric expressions of type INTEGER, REAL or double-precision REAL.

do-variable is a scalar variable of type INTEGER, REAL or double-precision REAL.

io-control-specs is a comma-separated list of
*[*UNIT =*] io-unit*
or *[*FMT =*] format*
or *[*NML =*] namelist-group-name*
or REC=*record*
or IOSTAT=*stat*
or ERR=*errlabel*
or END=*endlabel*
or EOR=*eorlabel*
or ADVANCE=*advance*
or SIZE=*size*

io-unit is an external file unit or *
format is a format specification (see *"Input/Output Editing"* beginning on page 25).

namelist-group-name is the name of a namelist group.

record is the number of the direct access record that is to be read.

stat is a scalar default INTEGER variable that is assigned a positive value if an error condition occurs, a negative value if an end-of-file or end-of-record condition occurs, and zero otherwise.

errlabel is a label that is branched to if an error condition occurs and no end-of-record condition or end-of-file condition occurs during execution of the statement.

endlabel is a label that is branched to if an end-of-file condition occurs and no error condition occurs during execution of the statement.

eorlabel is a label that is branched to if an end-of-record condition occurs and no error condition or end-of-file condition occurs during execution of the statement.

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.

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.

Remarks

io-control-specs must contain only one *io-unit*, and cannot contain both a *format* and a *namelist-group-name*.

A *namelist-group-name* must not appear if *inputs* is present.

If the optional characters UNIT= are omitted before *io-unit*, *io-unit* must be the first item in *io-control-specs*. If the optional characters FMT= are omitted before *format*, *format* must be the second item in *io-control-specs*. If the optional characters NML= are omitted before *namelist-group-name*, *namelist-group-name* must be the second item in *io-control-specs*.

If *io-unit* is an internal file, *io-control-specs* must not contain a REC= specifier or a *namelist-group-name*.

If the file is open for DIRECT, BINARY or TRANSPARENT access, an END= specifier must not appear, a *namelist-group-name* must not appear, and *format* must not be an asterisk indicating list-directed I/O.

An ADVANCE= specifier can appear only in formatted sequential I/O with an explicit format specification (*format-expr*) whose control list does not contain an internal file specifier. If an EOR= or SIZE= specifier is present, an ADVANCE= specifier must also appear with the value NO.

The *do-variable* of an *implied-do-control* that is contained within another *io-implied-do* must not appear as the *do-variable* of the containing *io-implied-do*.

Example

```
   character(len=30) :: intfile
   integer :: ios
   read *,a,b,c            ! read values from stdin
                           ! using list directed i/o
   read (3,"(3i10)") i,j,k ! read from unit 3 using format
   read 10,i,j,k  ! read stdin using format at label 10
10 format (3i10)
   read (11) a,b,c ! read unformatted data from unit 11
   intfile="         1         2         3"
   read(intfile,10) i,j,k       ! read from internal file
   read(12,rec=2) a,b,c         ! read direct access file
   read(13,10,err=20) i,j       ! read with error branch
20 read(13,10,iostat=ios) a     ! read with status return
   read(13,10,advance='no') i,j ! next read from same line
```

REAL Function

Description

The REAL function converts a number to a REAL data type.

Syntax

REAL (*a [, kind]*)

Required Arguments

a is an INTENT(IN) scalar or array of any numeric type.

Optional Arguments

kind is INTENT(IN) and determines the kind of the result. It must be a scalar INTEGER expression that can be evaluated at compile time. To maintain portability, this argument should be the result of a *"KIND Function"* or *"SELECTED_REAL_KIND Function"*.

Result

The result is of type REAL. Its value is a REAL representation of *a*

If *kind* is present, it determines the kind of the result.

If *a* is of type COMPLEX, the result's value is the real part of *a*.

Example

```
integer :: i=10.
real :: a=2.5
complex :: c=(1.5,2.5)
write(*,*) i,real(i)            ! convert integer to real
write(*,*) a,real(a,kind(1.d0)) ! convert between real kinds
write(*,*) c,real(c)            ! return real part
```

REAL Statement

Description

The REAL statement declares entities having the REAL data type.

Syntax

REAL *[kind-selector] [[, attribute-list] ::] entity [, entity] ...*

Where:

kind-selector is (*[*KIND=*] scalar-int-initialization-expr*)

scalar-int-initialization-expr is a scalar INTEGER expression that can be evaluated at compile time.

attribute-list is a comma-separated list from the following attributes: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), EXTERNAL, INTENT(IN) or INTENT(OUT) or INTENT(IN OUT), PUBLIC or PRIVATE, INTRINSIC, OPTIONAL, POINTER, SAVE, TARGET, VOLATILE, DLL_EXPORT or DLL_IMPORT or ML_EXTERNAL.

entity is *entity-name [(array-spec)] [=initialization-expr]*
or *function-name [(array-spec)]*

array-spec is an array specification.

initialization-expr is an expression that can be evaluated at compile time.

entity-name is the name of an entity being declared.

function-name is the name of a function being declared.

Remarks

function-name must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

The =*initialization-expr* must appear if the statement contains a PARAMETER attribute.

If =*initialization-expr* appears, a double colon must appear before the list of *entities*. Each *entity* has the SAVE attribute, unless it is in a named common block.

The =*initialization-expr* must not appear if *entity-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 pointer, an external name, an intrinsic name, or an automatic object.

If an array or function with an array result is declared with the POINTER attribute, it must have a deferred shape.

If an array is declared with the ALLOCATABLE attribute, it must have a deferred shape.

If an array or function with an array result does not have the POINTER or the ALLOCATABLE attribute, it must be specified with an explicit shape.

If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attributes must not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attributes must not be specified.

The PARAMETER attribute cannot be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

The INTENT(IN), INTENT(OUT), INTENT(IN OUT), and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* may not have the PUBLIC attribute if its type has the PRIVATE attribute.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

An *entity* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

A subprogram must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

An *entity* having the ALLOCATABLE attribute cannot be a dummy argument or a function result.

An array must not have both the ALLOCATABLE attribute and the POINTER attribute.

If an *entity* has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

An *entity* may not be given the same attribute more than once in a scoping unit.

Example

```
real :: a,b(2,4)               ! explicit shape
real,dimension(2) :: c=/1.,2./ ! has save attribute
real,pointer :: d(:)           ! deferred shapes
real,allocatable :: e(:)
real,parameter :: f=3          ! must be initialized
```

REPEAT Function

Description

The REPEAT function concatenates copies of a string.

Syntax

REPEAT (*string*, *ncopies*)

Arguments

string is an INTENT(IN) scalar of type CHARACTER

ncopies is an INTENT(IN) 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

```
write(*,*) repeat('ho',3) ! writes hohoho
```

RESHAPE Function

Description

The RESHAPE function constructs an array of a specified shape from a template array.

Syntax

RESHAPE (*source, shape [, pad] [, order]*)

Required Arguments

source is an INTENT(IN) array of any type. If *pad* is absent or of size zero, the size of *source* must be greater than or equal to the product of the values of the elements of *shape*.

shape is an INTENT(IN) INTEGER array of rank one. Its size must be positive and less than or equal to seven. It cannot have any negative elements.

Optional Arguments

pad is an INTENT(IN) array of the same type and kind as *source*.

order is an INTENT(IN) array of type INTEGER with the same shape as *shape*. Its value must be a permutation of (1, 2, ..., *n*), where *n* is the size of *order*. If *order* is absent, it is as if it were present with the value (1, 2, ..., *n*).

Result

The result is an array of the same type and kind as *source*, with a shape identical to *shape*.

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

```
real :: x1(4)
real :: x2(2,2)=reshape((/1.,2.,3.,4./),shape(x2))
real :: x3(3,2)
x1=reshape(x2,shape(x1))
write(*,*) x1 ! writes 1. 2. 3. 4.
write(*,*) reshape(x1,shape(x2),order=(/2,1/))
              ! writes 1. 3. 2. 4.
write(*,*) reshape(x1,shape(x3),pad=(/0./))
              ! writes 1. 2. 3. 4. 0. 0.
```

RETURN Statement

Description

The RETURN statement causes a transfer of control from a subprogram back to the calling procedure. Execution continues at the statement following the procedure invocation.

Syntax

RETURN *[alt-return]*

Where:
alt-return-label is a scalar INTEGER expression.

Remarks

If *alt-return* 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.

Example

```
subroutine zee()
    return         ! transfer of control back to caller
end subroutine zee
```

REWIND Statement

Description

The REWIND statement repositions a file to its initial point.

Syntax

REWIND (*position-spec-list*)

Where:
position-spec-list is *[[*UNIT *=] unit-number][*, ERR=*label][*, IOSTAT=*stat]* where UNIT=, ERR=, and IOSTAT= can be in any order but if UNIT= is omitted, then *unit-number* must be first.

unit-number is a scalar INTEGER expression corresponding to the input/output unit number of an external file.

label is a statement label that is branched to if an error condition occurs during execution of the statement.

stat is a variable of type INTEGER that is assigned a positive value if an error condition occurs, a negative value if an end-of-file or end-of-record condition occurs, and zero otherwise.

Remarks

Rewinding a file that is connected but has zero size has no effect.

Note that REWIND may only be used on sequential access files.

Example

```
integer :: ios
```

```
rewind 10              ! file on unit 10 rewound
rewind (10,iostat=ios) ! rewind with status
```

RRSPACING Function

Description

The RRSPACING function returns the reciprocal of relative spacing near a given number.

Syntax

RRSPACING (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is of the same type and kind as *x*. Its value is the reciprocal of the spacing; *x* divided by SPACING(*x*)

Example

```
real(kind(1.e0)) :: r10=1.e0
real(kind(1.d0)) :: r100=1.d0
real(kind(1.q0)) :: r1000=1.q0
write(*,*) r10/spacing(r10) ! writes 8388608.00
write(*,*) rrspacing(r10)   ! writes 8388608.00
write(*,*) rrspacing(r100)  ! writes 4503599627370496.
write(*,*) rrspacing(r1000)
         ! writes 5192296858534827628530496329220096.0
```

SAVE Statement

Description

The SAVE statement specifies that all data objects listed retain any previous association, allocation, definition, or value upon reentry of a subprogram.

Syntax

SAVE *[[*::*] saved-entities]*

Where:

saved-entities is a comma-separated list of *object-name*
or / *common-block-name* /

object-name is the name of a data object.

common-block-name is the name of a common block.

Remarks

Objects declared with the SAVE attribute in a subprogram are shared by all instances of the subprogram.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

A SAVE statement without a *saved-entities* list specifies that all allowable objects in the scoping unit have the SAVE attribute.

If a common block name appears in a SAVE statement other than in the main program, it must be have the SAVE attribute in every scoping unit in which the name appears.

A SAVE statement in a main program has no effect.

Example

```
subroutine sub1()
  logical,save :: first_time=.true. ! save attribute
  integer :: saveval
  save :: saveval                     ! save statement
  if(first_time) then                 ! do initializations
    first_time=.false.
    saveval=1
  end if
  saveval=saveval+1               ! value is preserved
end subroutine
```

SCALE Function

Description

The SCALE function multiplies a REAL number by a power of two.

Syntax

SCALE (*x*, *i*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

i is an INTENT(IN) scalar or array of type INTEGER.

If both *x* and *i* are arrays, they must have the same shape.

Result

The result is the same type and kind as *x*. Its value is $x \times 2^i$.

If either or both arguments are arrays, the result is an array of the same shape. Its values are as though the scalar SCALE operation were performed on each respective array element.

Example

```
real :: x=1.5,xa(2)=(/2.5,3.5/)
integer :: i=3, ia(2)=(/2,4/)
write(*,*) scale(x,i)   ! writes 12.0
write(*,*) scale(xa,i)  ! writes 20.0 28.0
write(*,*) scale(x,ia)  ! writes 6.0 24.0
write(*,*) scale(xa,ia) ! writes 10.0 56.0
```

SCAN Function

Description

The SCAN function scans a string for any one of a set of characters.

Syntax

SCAN (*string*, *set [*, *back]*)

Required Arguments

string is an INTENT(IN) scalar or array of type CHARACTER.

set is an INTENT(IN) scalar or array of type CHARACTER.

Optional Arguments

back is an INTENT(IN) scalar or array of type LOGICAL.

If more than one argument is an array, they must all have the same shape.

Result

The result is of type 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*.

If one or more arguments are arrays, the result is an array of the same shape. The value of each element of the resulting array is as if the scalar SCAN operation were performed on each respective element of the input arrays.

Example

```
character(len=12) :: c1="Howdy there!"
character(len=6) :: c2(2)=(/"Howdy ","there!"/)
character(len=3) :: c3(2)=(/"def","ghi"/)
write(*,*) scan(c1,'def')                ! writes 4
write(*,*) scan(c2,c3)                   ! writes 4 2
write(*,*) scan(c1,'def',back=.true.)    ! writes 11
write(*,*) scan(c2,c3,(/.true.,.false./)) ! writes 4 2
```

SELECTED_INT_KIND Function

Description

The SELECTED_INT_KIND function returns the kind type parameter of an INTEGER data type.

Syntax

SELECTED_INT_KIND (*r*)

Arguments

r is an INTENT(IN) scalar INTEGER.

Result

The result is a scalar default INTEGER. Its value is equal to the processor dependent kind type parameter of the data type that accommodates all values *n* with $-10^r < n < 10^r$.

If more than one kind is available, the return value is the kind type parameter with the smaller decimal exponent range.

If no such kind is available in the specified range, the result is -1.

Example

```
write(*,*) selected_int_kind(2)  ! writes 1
write(*,*) selected_int_kind(4)  ! writes 2
write(*,*) selected_int_kind(7)  ! writes 4
write(*,*) selected_int_kind(12) ! writes 8
write(*,*) selected_int_kind(20) ! writes -1
```

SELECTED_REAL_KIND Function

Description

The SELECTED_REAL_KIND function returns the 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*.

Syntax

SELECTED_REAL_KIND (*[p] [, r]*)

Optional Arguments

p is an INTENT(IN) scalar INTEGER, representing the requested precision.

r is an INTENT(IN) scalar INTEGER representing the requested exponent range.

At least one argument must be present.

Result

The result is a scalar default INTEGER. Its value is equal to the processor dependent 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 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.

The result is -1 if the precision is not available, -2 if the range is not available, and -3 if neither is available.

Example

```
! request a precision
write(*,*) selected_real_kind(p=6)  ! writes 4
write(*,*) selected_real_kind(p=12) ! writes 8
write(*,*) selected_real_kind(p=24) ! writes 16
write(*,*) selected_real_kind(p=48) ! writes -1
! request a range
write(*,*) selected_real_kind(r=10)    ! writes 4
write(*,*) selected_real_kind(r=100)   ! writes 8
write(*,*) selected_real_kind(r=1000)  ! writes 16
write(*,*) selected_real_kind(r=10000) ! writes -2
write(*,*) selected_real_kind(r=10000,p=48) ! writes -3
```

SEQUENCE Statement

Description

The SEQUENCE statement specifies a storage sequence for objects of a derived type. It can only appear within a 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 must be sequence types.

Example

```
type zee
  sequence      ! zee is a sequence type
  real :: a,b,c ! a,b,c is the storage sequence
end type zee
```

SET_EXPONENT Function

Description

The SET_EXPONENT function returns the model representation of a number with the exponent part set to a power of two.

Syntax

SET_EXPONENT (*x*, *i*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

i is an INTENT(IN) scalar or array of type INTEGER.

If both arguments are arrays, they must have the same shape.

Result

The result is of the same type and kind as *x*. Its value is FRACTION(*x*)*2^i.

If either or both arguments are arrays, the result is an array with the same shape. The value of each result element is as though the scalar SET_EXPONENT operation were performed for each respective element of the input arrays.

Example

```
real :: x=4.3,xa(2)=(/1.5,2.5/)
integer :: i=2,ia(2)=(/4,5/)
write(*,*) fraction(x)*2**i    ! writes 2.15
write(*,*) set_exponent(x,i)   ! writes 2.15
write(*,*) set_exponent(xa,i)  ! writes 3.0 2.5
write(*,*) set_exponent(x,ia)  ! writes 8.6 17.2
write(*,*) set_exponent(xa,ia) ! writes 12.0 20.0
```

SHAPE Function

Description

The SHAPE function returns the shape of an array argument.

Syntax

SHAPE (*source*)

Arguments

source is an INTENT(IN) scalar or array of any type.

source must not be an assumed-size array, a pointer that is disassociated or an allocatable array that is not allocated.

Result

The result is a default INTEGER array of rank one whose size is the rank of *source* and whose value is the shape of *source*.

If *source* is scalar, the result is an array of rank one and zero size.

Example

```
integer :: i,ia(-2:2),ib(3,5,7),ic(9,2,4,6,5,3,3)
write(*,*) shape(i)  ! zero sized array
write(*,*) shape(ia) ! writes 5
write(*,*) shape(ib) ! writes 3 5 7
write(*,*) shape(ic) ! writes 9 2 4 6 5 3 3
```

SIGN Function

Description
The SIGN function transfers the sign of a REAL or INTEGER argument.

Syntax
SIGN (*a*, *b*)

Arguments
a is an INTENT(IN) scalar or array of type INTEGER or REAL.

b is an INTENT(IN) scalar or array of type INTEGER or REAL.

If both *a* and *b* are arrays, they must have the same shape.

Result
The result is of the same type and kind as *a*.

Its value is |a|, if b is greater than or equal to zero; and -|a| if b is less than zero. The compiler does not distinguish between positive and negative zero.

If either or both arguments are arrays, the result is an array with the same shape. Each element of the result is as though the scalar SIGN operation were performed on each respective element of the argument arrays.

Example
```
real :: r=1.
integer :: ia(2)=(/2,-3/)
write(*,*) sign(r,-1)  ! writes -1.
write(*,*) sign(r,ia)  ! writes 1., -1.
write(*,*) sign(ia,-1) ! writes -2 -3
```

SIN Function

Description
The SIN function returns the trigonometric sine of a REAL or COMPLEX argument.

Syntax
SIN (*x*)

Arguments
x is an INTENT(IN) scalar or array of type REAL or COMPLEX and must be expressed in radians.

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*.

Example

```
real :: x=.5,y(2)=(/1.,1./)
complex :: z=(1.,1.)
write(*,*) sin(x) ! writes .4794255
write(*,*) sin(y) ! writes .8414709 .8414709
write(*,*) sin(z) ! writes (1.298457 .6349639)
```

SINH Function

Description

The SINH function returns the hyperbolic sine of a REAL argument.

Syntax

SINH (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is of the same type and kind as *x*. Its value is a REAL representation of the hyperbolic sine of *x*.

Example

```
real :: x=.5,y(2)=(/1.,1./)
write(*,*) sinh(x) ! writes .5210953
write(*,*) sinh(y) ! writes 1.175201 1.175201
```

SIZE Function

Description

The SIZE function returns the size of an array or a dimension of an array.

Syntax

SIZE (*array [, dim]*)

Required Arguments

array is an INTENT(IN) array of any type. It must not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Arguments

dim is an INTENT(IN) scalar of type INTEGER and must be a dimension of *array*. If *array* is assumed-size, *dim* must be present and less than the rank of *array*

Result

The result is a scalar default INTEGER.

If *dim* is present, the result is the extent of dimension *dim* of *array*.

If *dim* is absent, the result is the total number of elements in *array*.

Example

```
integer,dimension(3,-4:0) :: i
integer :: k,j
write(*,*) size (i)   ! writes 15
write(*,*) size (i,2) ! writes 5
```

SPACING Function

Description

The SPACING function returns the absolute spacing near a given number; the smallest number that can be added to the argument to produce a number that is different than the argument.

Syntax

SPACING (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is of the same type and kind as *x*. Its value is the spacing of REAL values near *x*.

Example

```
real :: a=1.,b=1.e10
real(kind(1.d0)) :: c=1.d0,d=1.d10
write(*,*) nearest(a,1.)-a ! writes 1.1920929
write(*,*) epsilon(a)      ! writes 1.1920929
write(*,*) spacing(a)      ! writes 1.1920929
write(*,*) spacing(b)      ! writes 1024.0000
write(*,*) spacing(c)      ! writes 2.22044604925031e-16
write(*,*) spacing(d)      ! writes 1.90734863281250e-06
```

SPREAD Function

Description

The SPREAD function adds a dimension to an array by adding copies of a data object along a given dimension.

Syntax

SPREAD (*source*, *dim*, *ncopies*)

Arguments

source is an INTENT(IN) scalar or array of any type. Its rank must be less than seven.

dim is an INTENT(IN) scalar of type INTEGER with a value in the range $1 \le dim \le n + 1$, where *n* is the rank of *source*.

ncopies is an INTENT(IN) 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 an array 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 :: b(2,2)=reshape((/1,2,3,4/),shape(b))
! show how shape of array changes after spreading
write(*,*) shape(b)             ! writes 2 2
write(*,*) shape(spread(b,1,3)) ! writes 3 2 2
write(*,*) shape(spread(b,2,3)) ! writes 2 3 2
write(*,*) shape(spread(b,3,3)) ! writes 2 2 3
! show element values after spreading
write(*,*) b            ! writes 1 2 3 4
write(*,*) spread(b,1,3) ! writes 1 1 1 2 2 2 3 3 3 4 4 4
write(*,*) spread(b,2,3) ! writes 1 2 1 2 1 2 3 4 3 4 3 4
write(*,*) spread(b,3,3) ! writes 1 2 3 4 1 2 3 4 1 2 3 4
```

SQRT Function

Description

The SQRT function returns the square root of a REAL or COMPLEX argument.

Syntax

SQRT (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL or COMPLEX.

If *x* is REAL, its value must be greater than or equal to zero.

Result

The result is the same kind and type as *x*.

If *x* is REAL, the result value is a REAL representation of the square root of *x*.

If *x* is 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.

Example

```
real :: x1=4.,x2(2)=(/2.,6./)
complex :: q=(-1.,0.)
write(*,*) sqrt(x1) ! writes 2.0
write(*,*) sqrt(x2) ! writes 1.4142135 2.4494898
write(*,*) sqrt(q)  ! writes (0., 1.)
```

Statement Function

Description

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

Syntax

function-name (*[dummy-args]*)=*scalar-expr*

Where:

function-name is the name of the function being defined.

dummy-args 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, scalar variables, array elements, references to functions and function dummy procedures, and intrinsic operators.

If a reference to a statement function appears in *scalar-expr*, its definition must have been provided earlier in the scoping unit and must not be the name of the statement function being defined.

Each scalar variable reference in *scalar-expr* must 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 must not be supplied as a procedure argument.

Example

```
mean(a,b)=(a+b)/2
c=mean(2.0,3.0) ! c is assigned the value 2.5
```

STOP Statement

Description

The STOP statement causes execution of a program to terminate.

Syntax

STOP *[stop-code]*

Where:

stop-code is a scalar CHARACTER constant or a series of 1 to 5 digits.

Remarks

When a STOP statement is reached, the optional *stop-code* is displayed, if present.

Example

```
program foo
  stop      ! program execution terminated
end program foo
```

SUBROUTINE Statement

Description

The SUBROUTINE statement begins a subroutine subprogram. It specifies the subroutines name and dummy arguments, and any special characteristics such as PURE, ELEMENTAL, or RECURSIVE.

Syntax

*[*PURE*] [*ELEMENTAL*] [*RECURSIVE*]* SUBROUTINE *subroutine-name* (*[dummy-arg-names]*)

Where:

subroutine-name is the name of the subroutine.

dummy-arg-names is a comma-separated list of dummy argument names.

Remarks

The prefixes PURE, ELEMENTAL, and RECURSIVE may appear in any order.

A subroutine with the prefix PURE or ELEMENTAL is subject to the additional constraints of pure procedures, which ensure that no unseen side effects occur on invocation of the subroutine. See “PURE Procedure” on page 225.

An ELEMENTAL subroutine is subject to the constraints of elemental procedures. See “ELEMENTAL Procedure” on page 123.

The keyword RECURSIVE must be present if the subroutine directly or indirectly calls itself or a subroutine defined by an ENTRY statement in the same subprogram. RECURSIVE must also be present if a subroutine defined by an ENTRY statement directly or indirectly calls itself, another subroutine defined by an ENTRY statement, or the subroutine defined by the SUBROUTINE statement.

Example

```
subroutine sub1() ! subroutine statement with no arguments
      common /c1/ a
      a=1.
end subroutine
subroutine sub2(a,b,c) ! subroutine statement with arguments
      real :: a,b,c
      a=b+c
end subroutine
recursive subroutine sub3(i) ! recursive required if the
    i=i-1                    ! subroutine calls itself
    if(i > 0) call sub3(i)   ! directly or indirectly
end subroutine
```

SUM Function

Description

The SUM function returns the sum of elements of an array, along a given dimension, for which a mask is true.

Syntax

SUM (*array [, dim] [, mask]*)

Required Arguments

array is an INTENT(IN) array of type INTEGER, REAL, or COMPLEX.

Optional Arguments

dim is an INTENT(IN) scalar INTEGER in the range $1 \le dim \le n$, where *n* is the rank of *array*. The corresponding dummy argument must not be an optional dummy argument.

mask is an INTENT(IN) scalar or array of type LOGICAL. It must be conformable with *array*.

Result

The result is of the same type and kind as *array*.

The result is scalar if *dim* is absent or if *array* has rank one; otherwise it is an array of rank *n*-1 and of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of *array*.

If *dim* is absent, the result is the sum of the values of all the elements of *array*.

If *dim* is present, the result is the sum of the values of all elements of *array* along dimension *dim*.

If *mask* is present, only the elements of *array* for which *mask* is true are considered.

Example

```
integer :: m(2,2)=reshape((/1,2,3,4/),shape(m))
write(*,'(2i3)') m          ! writes 1  2
                            !        3  4
write(*,*) sum(m)           ! writes 10
write(*,*) sum(m,dim=1)     ! writes 3 7
write(*,*) sum(m,dim=2)     ! writes 4 6
write(*,*) sum(m,mask=m>2) ! writes 7
```

SYSTEM Function (Linux only)

Description

The SYSTEM function executes a system command as if from the command line.

Syntax

SYSTEM (*cmd*)

Arguments

cmd is an INTENT(IN) scalar of type CHARACTER. It contains the system command to be executed as if it were typed on the command line.

Result

The result is of type INTEGER. It is the exit status of the system command.

Example

```
if (system("ls > current.dir") /= 0) write(*,*) "Error"
! puts a listing of the current directory into
! the file 'current.dir'
```

SYSTEM Subroutine

Description

The SYSTEM subroutine executes a system command as if from the command line.

Syntax (Windows)

SYSTEM (*cmd [, dosbox] [, spawn]*)

Syntax (Linux)

SYSTEM (*cmd*)

Required Arguments

cmd is an INTENT(IN) scalar of type CHARACTER. It is the system command to be executed as if it were typed on the command line.

Optional Arguments

dosbox is an INTENT(IN) scalar of type LOGICAL. It has the value true if a new DOS box is to be opened (required for internal commands like DIR) and false otherwise. By default, *dosbox* has the value true.

spawn is an INTENT(IN) scalar of type LOGICAL. It has the value true if the command or program to be executed is to be spawned as a separate process and false otherwise. By default, spawn has the value true.

Example

```
call system("dir > current.dir")
! puts a listing of the current directory into
! the file 'current.dir'
```

SYSTEM_CLOCK Subroutine

Description

The SYSTEM_CLOCK subroutine returns INTEGER data from the real-time clock.

Syntax

SYSTEM_CLOCK (*[count] [, count_rate] [, count_max]*)

Optional Arguments

count is an INTENT(OUT) scalar of type default INTEGER. Its value is set to the current value of the processor clock or to -HUGE(0) if no clock is available.

count_rate is an INTENT(OUT) scalar of type default INTEGER. It is set to the number of processor clock counts per second, or to zero if there is no clock.

count_max is an INTENT(OUT) scalar of type default INTEGER. It is set to the maximum value that *count* can have, or zero if there is no clock.

At least one argument must be present.

Example

```
integer :: c,cr,cm
call system_clock(c,cr,cm)
write(*,*) c                 ! writes current count
write(*,*) cr                ! writes count rate
write(*,*) cm                ! writes maximum count possible
write(*,*) real(c)/real(cr) ! current count in seconds
```

TAN Function

Description

The TAN function returns the trigonometric tangent of a REAL argument.

Syntax

TAN (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL, and must be expressed in radians.

Result

The result is of the same type and kind as *x*. Its value is a REAL representation of the tangent of *x*.

Example

```
real :: x=.5,y(2)=(/1.,1./)
write(*,*) tan(x) ! writes .54630249
write(*,*) tan(y) ! writes 1.5574077 1.5574077
```

TANH Function

Description

The TANH function returns the hyperbolic tangent of a REAL argument.

Syntax

TANH (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

Result

The result is of the same type and kind as *x*. Its value is a REAL representation of the hyperbolic tangent of *x*.

Example

```
real :: x=.5,y(2)=(/1.,1./)
write(*,*) tanh(x) ! writes .4621171
write(*,*) tanh(y) ! writes .7615941 .7615941
```

TARGET Statement

Description

The TARGET statement specifies that data objects have the target attribute and thus can be associated with a pointer.

Syntax

TARGET *[::] object-name [(array-spec)] [, object-name [(array-spec)]] ...*

Where:

object-name is the name of a data object.

array-spec is an array specification.

Example

```
integer,pointer :: z
integer :: a=1
target :: a                    ! target statement
integer,target :: b=2,c=3 ! target attribute
z => a
write(*,*) z
z => b
write(*,*) z
z => c
write(*,*) z
```

TINY Function

Description

The TINY function returns the smallest positive number of a numeric data type that can be represented without loss of precision.

Syntax

TINY (*x*)

Arguments

x is an INTENT(IN) scalar or array of type REAL.

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*.

Example

```
real(kind(1.e0)) :: r10
real(kind(1.d0)) :: r100
real(kind(1.q0)) :: r1000
write(*,*) tiny(r10)   ! writes 1.1754943E-38
write(*,*) tiny(r100)  ! writes 2.2250738585072E-308
write(*,*) tiny(r1000)
      ! writes 3.3621031431120935062626778173217552E-4932
```

TRANSFER Function

Description

The TRANSFER function interprets the physical representation of a number with the type and type parameters of a given number.

Syntax

TRANSFER (*source*, *mold [, size]*)

Required Arguments

source is an INTENT(IN) scalar or array of any type.

mold is an INTENT(IN) scalar or array of any type.

Optional Arguments

size is an INTENT(IN) scalar of type INTEGER. The corresponding actual argument must not be a optional dummy argument.

Result

The result is of the same type and type parameters as *mold*.

If *mold* is scalar and *size* is absent the result is a scalar.

If *mold* is an array and *size* is absent, the result is an array 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 an array 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

```
character(len=4) :: c="LOVE"
integer :: i,j(2,2)
real :: r
logical :: l
write(*,*) transfer(c,i) ! writes 1163284300
write(*,*) transfer(c,r) ! writes 3428.95605
write(*,*) transfer(c,l) ! writes T
```

TRANSPOSE Function

Description

The TRANSPOSE function transposes an array of rank two.

Syntax

TRANSPOSE (*matrix*)

Arguments

matrix is an INTENT(IN) rank two array of any type.

Result

The result is of rank two and the same type and kind 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:: a(2,3)=reshape((/1,2,3,4,5,6/),shape(a))
write(*,'(2i3)') a             ! writes 1 2
                               !        3 4
                               !        5 6
write(*,*) shape(a)            ! writes 2 3
write(*,'(3i3)') transpose(a)  ! writes 1 3 5
                               !        2 4 6
write(*,*) shape(transpose(a)) ! writes 3 2
```

TRIM Function

Description

The TRIM function omits trailing blanks from a character argument.

Syntax

TRIM (*string*)

Arguments

string is an INTENT(IN) scalar of type CHARACTER.

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=10) :: c="Howdy!    "
write(*,*) len(c)        ! writes 10
write(*,*) c,'end'       ! writes Howdy!    end
write(*,*) len(trim(c))  ! writes 6
write(*,*) trim(c),'end' ! writes Howdy!end
```

Type Declaration Statement

See INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, or TYPE statement.

TYPE Statement

Description

The TYPE statement defines a derived type, and declares entities having a derived type.

Syntax

(Definition)

TYPE *[[, access-spec]* ::*] type-name*

or

(Declaration)

TYPE (*type-name*) *[, attribute-list* ::*] entity [, entity]* ...

Where:

access-spec is PUBLIC
or PRIVATE

type-name is the name of the derived type being defined.

attribute-list is a comma-separated list from the following attributes: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), EXTERNAL, INTENT(IN) or INTENT(OUT) or INTENT(IN OUT), PUBLIC or PRIVATE, INTRINSIC, OPTIONAL, POINTER, SAVE, TARGET, VOLATILE, DLL_EXPORT or DLL_IMPORT or ML_EXTERNAL.

entity is *entity-name [*(*array-spec*)*] [=initialization-expr]*
or *function-name [*(*array-spec*)*]*

array-spec is an array specification.

initialization-expr is an expression that can be evaluated at compile time.

entity-name is the name of an entity being declared.

function-name is the name of a function being declared.

Remarks

access-spec is permitted only if the derived type definition is within the specification part of a module.

If a component of a derived type is of a type declared to be private, either the definition must contain the PRIVATE statement or the derived type must be private.

type-name must not be the name of an intrinsic type nor of another accessible derived type name.

function-name must be the name of an external, intrinsic, or statement function, or a function dummy procedure.

The =*initialization-expr* must appear if the statement contains a PARAMETER attribute.

If =*initialization-expr* appears, a double colon must appear before the list of *entities*. Each *entity* has the SAVE attribute, unless it is in a named common block.

The =*initialization-expr* must not appear if *entity-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 pointer, an external name, an intrinsic name, or an automatic object.

If an array or function with an array result is declared with the POINTER attribute, it must have a deferred shape.

If an array is declared with the ALLOCATABLE attribute, it must have a deferred shape.

If an array or function with an array result does not have the POINTER or the ALLOCATABLE attribute, it must be specified with an explicit shape.

If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attributes must not be specified.

If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attributes must not be specified.

The PARAMETER attribute cannot be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

The INTENT(IN), INTENT(OUT), INTENT(IN OUT), and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* may not have the PUBLIC attribute if its type has the PRIVATE attribute.

The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, or an automatic data object.

An *entity* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

A subprogram must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

An *entity* having the ALLOCATABLE attribute cannot be a dummy argument or a function result.

An array must not have both the ALLOCATABLE attribute and the POINTER attribute.

If an *entity* has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

An *entity* may not be given the same attribute more than once in a scoping unit.

Example

```
type zee   ! type definition
  sequence
  real :: a,b
  integer :: i
end type zee
type (zee) :: a,b,c(2,2)       ! type declaration
type (zee) :: e=zee(2.,3.5,-1) ! with initialization
```

UBOUND Function

Description

The UBOUND function retrieves the upper bounds of an array or a dimension of an array.

Syntax

UBOUND (*array [, dim]*)

Required Arguments

array is an INTENT(IN) array of any type. It must not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Arguments

dim is an INTENT(IN) scalar of type INTEGER and must be a dimension of *array*.

Result

The result is of type 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)
write(*,*) ubound(j)   ! writes 2
write(*,*) ubound(i)   ! writes 3 0
write(*,*) ubound(i,2) ! writes 0
write(*,*) ubound(i,1) ! writes 3
```

UNDFL Subroutine (Windows Only)

Description

The UNDFL subroutine masks and detects floating-point underflow exceptions.

Syntax

UNDFL (*lflag*)

Arguments

lflag is an INTENT(IN) scalar of type LOGICAL. It is assigned the value true if an underflow exception has occurred, and false otherwise.

Remarks

lflag must be set to true on the first invocation.

The initial invocation of the UNDFL subroutine masks the underflow interrupt on the floating-point unit.

Subsequent invocation returns an *lflag* value of true if the exception has occurred or false if the exception has not occurred.

Example

```
real(kind(1.d0)) :: a=tiny(a)
```

```
logical :: lflag = .true.
call undfl(lflag)  ! mask the underflow interrupt
write(*,*) lflag   ! writes F
do
  a=a/2.d0
  call undfl(lflag)! test for underflow
  if(lflag) exit
end do
write(*,*) lflag   ! writes T
```

UNPACK Function

Description

The UNPACK function unpacks an array of rank one into an array under control of a mask.

Syntax

UNPACK (*vector*, *mask*, *field*)

Arguments

vector is an INTENT(IN) rank one array of any type. Its size must be at least as large as the number of true elements in *mask*.

mask is an INTENT(IN) array of type LOGICAL.

field must be of the same type and type parameters as *vector*. It must 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./),shape(d))
e=unpack(c,mask=d,field=-1)
write(*,'(2i3)') e ! writes -1  0
                   !         3 -1
```

USE Statement

Description

The USE statement specifies that a module is accessible from the current scoping unit. It also provides a means of renaming or limiting the accessibility of entities in the module.

Syntax

USE *module [, rename-list]*

or

USE *module,* ONLY: *[only-list]*

Where:

module is the name of a module.

rename-list is a comma-separated list of *local-name* => *use-name*

only-list is a comma-separated list of *access-id*
or *[local-name* => *use-name]*

local-name is the local name for the entity specified by *use-name*

use-name is the name of a public entity in the specified module

access-id is *use-name*
or *generic-spec*

generic-spec is *generic-name*
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

generic-name is the name of a generic procedure.

defined-operator is one of the intrinsic operators
or *.op-name*.

op-name is a user-defined name for the operation.

Remarks

A USE statement without ONLY provides access to all PUBLIC entities in the specified module.

A USE statement with ONLY provides access only to those entities that appear in the *only-list*.

If more than one USE statement appears in a scoping unit, the *rename-list*s and *only-list*s are treated as one concatenated *rename-list*.

If two or more generic interfaces 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.

Two or more accessible entities, other than generic interfaces, can have the same name only if no entity is referenced by this name in the scoping unit.

If *local-name* is absent, the *use-name* is available by use association.

An entity can be accessed by more than one *local-name*.

A *local-name* must not be declared with different 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.

Forward references to modules are not allowed in LF95. That is, if a module is used in the same source file in which it resides, the module program unit must appear before its use.

Example

```
module mod1
      integer :: i,j,k
      real :: a,b,c
end module mod1
subroutine sub1()
  use mod1 ! a,b,c,i,j,k all available by use association
end subroutine sub1
subroutine sub2()
 use mod1, only: a,b ! a,b are available, c,i,j,k not available
end subroutine sub2
subroutine sub3()
  use mod1, aa=>a ! a is known as aa within the scope of sub3
end subroutine sub3
```

VAL Function

Description

The VAL function passes an item to a procedure by value. VAL is only used as an actual argument. The VAL function has largely been superceded by the CARG function.

Syntax

VAL (*item*)

Arguments

item is an INTENT(IN) data object of type INTEGER, REAL, or LOGICAL. It is the data object for which to return a value.

Result

The result is the value of *item*. Its C data type is as follows:

Table 11: VAL result types

Fortran Type	Fortran Kind	C type
INTEGER	1	long int
INTEGER	2	long int
INTEGER	4	long int
REAL	4	float
COMPLEX	4	must not be passed by value; if passed by reference (without CARG) it is a pointer to a structure of the form: struct complex { float real_part; float imaginary_part;};
LOGICAL	1	unsigned long
LOGICAL	4	unsigned long
CHARACTER	1	must not be passed by value with VAL

Example

```
i=my_c_function(val(a)) ! a is passed by value
```

VALUE Statement

Description

The VALUE statement specifies that the dummy argument is passed by value, rather than by reference.

Syntax

[type-decl,] VALUE *[::] var*

WHERE:

type-decl is an intrinsic or derived type data declaration

var is a variable name

Remarks

The VALUE statement may only be specified for a dummy argument.

If the VALUE statement is specified, the PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, INTENT(INOUT), or INTENT(OUT) attributes cannot be specified for that variable.

If the VALUE statement is specified for a dummy argument of type CHARACTER, the length parameter shall be omitted, or be specified by an initialization expression having a value of one.

If a dummy argument has the VALUE attribute, a temporary copy of the actual argument is made, and the copy is associated with the dummy argument. Subsequent changes to the dummy argument do not affect the value or status of the actual argument.

If the dummy argument has both the VALUE and TARGET attributes, any pointers associated with the dummy argument become undefined when execution of the procedure is complete.

By default, Fortran passes arguments by reference.

Example

```
subroutine method1(valuearg1,valuearg2)
   real, value :: valuearg1 ! value attribute
   integer :: valuearg2
   value   :: valuearg2      ! value statement
   ! do something
end subroutine
```

VERIFY Function

Description

The VERIFY function verifies that a set of characters contain all the characters in a string.

Syntax

VERIFY (*string*, *set [*, *back]*)

Required Arguments

string is an INTENT(IN) scalar or array of type CHARACTER.

set is an INTENT(IN) scalar or array of type CHARACTER

Optional Arguments

back is an INTENT(IN) scalar or array of type LOGICAL.

If any or all the arguments are arrays, they must all have the same shape.

Result

The result is of type 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.

If one or more arguments are arrays, the result is an array of the same shape. The value of each element of the resulting array is as if the scalar SCAN operation were performed on each respective element of the input arrays.

Example

```
character(len=12) :: c1="Howdy there!"
character(len=6) :: c2(2)=(/"Howdy ","there!"/)
character(len=2) :: c3(2)=(/"de","gh"/)
write(*,*) verify(c1,'de')                 ! writes 1
write(*,*) verify(c2,c3)                   ! writes 1 1
write(*,*) verify(c1,'de',back=.true.)     ! writes 12
write(*,*) verify(c2,c3,(/.true.,.false./)) ! writes 6 1
```

VOLATILE Statement

Description

The VOLATILE statement indicates that a data object may be referenced, become redefined or undefined by means not specified in the Fortran standard.

Syntax

VOLATILE *[::] object-name-list*

Where:

object-name-list is a list of data objects.

Remarks

If an object has the VOLATILE attribute, it cannot have the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attributes.

If an object has the VOLATILE attribute, then all of its subobjects are VOLATILE.

An object may have the VOLATILE attribute in one scoping unit without necessarily having it in another scoping unit.

If both POINTER and VOLATILE are specified, the volatility applies to the target of the POINTER and to the pointer association status.

If both ALLOCATABLE and VOLATILE are specified, the volatility applies to the allocation status, bounds and definition status.

Example

```
real :: r1
volatile :: r1      ! volatile statement
real,volatile :: r2 ! volatile attribute
```

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.

The WHERE statement signals the beginning of a WHERE construct.

The ELSE WHERE statement controls assignment of each element of a WHERE statement's logical mask that evaluates to false, and each element of the ELSE WHERE's logical mask that evaluates to true. It executes a block of assignment statements for each of the corresponding elements in an assignment expression.

The END WHERE statement signals the end of the innermost nested WHERE construct.

Syntax

```
WHERE (mask-expr)
        [assignment-stmt]
        [assignment-stmt]
        ...
[ELSEWHERE (mask-expr)]
        [assignment-stmt]
        [assignment-stmt]
        ...
[ELSE WHERE]
        [assignment-stmt]
        [assignment-stmt]
        ...
END WHERE
```

Where:

mask-expr is a LOGICAL expression.

assignment-stmt is an assignment statement.

Remarks

mask-expr is evaluated at the beginning of the masked array assignment and the result value governs the masking of assignments in the WHERE statement or construct. Subsequent changes to entities in *mask-expr* have no effect on the masking.

The variable on the left-hand side of *assignment-stmt* must have the same shape as *mask-expr*.

When *assignment-stmt* is executed, the right-hand side of the assignment is evaluated for all elements where *mask-expr* is true and the result assigned to the corresponding elements of the left-hand side.

If a non-elemental function reference occurs in the right-hand side of *assignment-stmt*, the function is evaluated without any masked control by the *mask-expr*.

assignment-stmt must not be a defined assignment statement.

Each statement in a WHERE construct is executed in sequence.

If the ELSE WHERE statement does not have a mask expression, it must be the last block of assignment code to appear in the construct.

There can be multiple ELSEWHERE statements with *mask-expr*s.

Example 1

```
integer :: a(3)=(/1,2,3/)
where (a == 2)
  a=-1
end where
```

Example 2

```
integer :: a(3)=(/1,2,3/),b(3)=(/3,2,1/)
where (b > a)
  a=b              ! a is assigned (/3,2,3/)
else where(b == a) ! (.NOT. b>a) .AND. b==a
  b=0              ! b is assigned (/3,0,1/)
elsewhere(a == 2)  ! (.NOT.b>a).AND.(.NOT.b==a).AND.a==2
  a=a+1            ! b==a got to these elements first
elsewhere   ! (.NOT. b>a) .AND.(.NOT.b==a).AND. (.NOT.a==2)
  b=-1      ! b is assigned (/3,0,-1/)
end where
write(*,*) a,b
```

WHERE Statement

Description

The WHERE statement masks the assignment of values in array assignment statements. The WHERE statement can begin a WHERE construct that contains zero or more assignment statements, or can itself contain an assignment statement.

Syntax

WHERE (*mask-expr*) *[assignment-stmt]*

Where:

mask-expr is a LOGICAL expression.

assignment-stmt is an assignment statement.

Remarks

If the WHERE statement contains no *assignment-stmt*, it specifies the beginning of a WHERE construct.

The variable on the left-hand side of *assignment-stmt* must have the same shape as *mask-expr*.

When *assignment-stmt* is executed, the right-hand side of the assignment is evaluated for all elements where *mask-expr* is true and the result assigned to the corresponding elements of the left-hand side.

If a non-elemental function reference occurs in the right-hand side of *assignment-stmt*, the function is evaluated without any masked control by the *mask-expr*.

mask-expr is evaluated at the beginning of the masked array assignment and the result value governs the masking of assignments in the WHERE statement or construct. Subsequent changes to entities in *mask-expr* have no effect on the masking.

assignment-stmt must not be a defined assignment.

Example

```
! a, b, and c are arrays
where (a>b) a=-1 ! where statement
where (b>c)      ! begin where construct
  b=-1
elsewhere
  b=1
end where
```

WRITE Statement

Description

The WRITE statement transfers values to an input/output unit from entities specified in an output list or a namelist group.

Syntax

WRITE *(io-control-specs*) *[outputs]*

Where:

outputs is a comma-separated list of *expr*
or *io-implied-do*

expr is a variable.

io-implied-do is (*outputs*, *implied-do-control*)

implied-do-control is *do-variable*=*start*, *end [*, *increment]*

start, *end*, and *increment* are scalar numeric expressions of type INTEGER, REAL or double-precision REAL.

do-variable is a scalar variable of type INTEGER, REAL or double-precision REAL.

io-control-specs is a comma-separated list of
*[*UNIT =*] io-unit*
or *[*FMT =*] format*
or *[*NML =*] namelist-group-name*
or REC=*record*
or IOSTAT=*stat*
or ERR=*errlabel*
or END=*endlabel*
or EOR=*eorlabel*
or ADVANCE=*advance*
or SIZE=*size*

io-unit is an external file unit, or *

format is a format specification (see *"Input/Output Editing"* beginning on page 25).

namelist-group-name is the name of a namelist group.

record is the number of the direct-access record that is to be written.

stat is a scalar default INTEGER variable that is assigned a positive value if an error condition occurs and zero otherwise.

errlabel is a label that is branched to if an error condition occurs and no end-of-record condition or end-of-file condition occurs during execution of the statement.

endlabel is a label that is branched to if an end-of-file condition occurs and no error condition occurs during execution of the statement.

eorlabel is a label that is branched to if an end-of-record condition occurs and no error condition or end-of-file condition occurs during execution of the statement.

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.

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.

Remarks

If the optional characters UNIT= are omitted before *io-unit, io-unit* must be the first item in *io-control-specs*. If the optional characters FMT= are omitted before *format, format* must be the second item in *io-control-specs*. If the optional characters NML= are omitted before *namelist-group-name, namelist-group-name* must be the second item in *io-control-specs*.

io-control-specs must contain exactly one *io-unit*, and must not contain both a *format* and a *namelist-group-name*.

A *namelist-group-name* must not appear if *outputs* is present.

If *io-unit* is an internal file, *io-control-specs* must not contain a REC= specifier or a *namelist-group-name*.

If the file is open for DIRECT, BINARY or TRANSPARENT access, an END= specifier must not appear, a *namelist-group-name* must not appear, and *format* must not be an asterisk indicating list-directed I/O.

An ADVANCE= specifier can appear only in formatted sequential I/O with an explicit format specification (*format-expr*) whose control list does not contain an internal file specifier. If an EOR= or SIZE= specifier is present, an ADVANCE= specifier must also appear with the value NO.

The *do-variable* of an *implied-do-control* that is contained within another *io-implied-do* must not appear as the *do-variable* of the containing *io-implied-do.*

If an array appears as an output item, it is treated as if the elements were specified in array-element order.

If a derived type object appears as an output item, it is treated as if all of the components were specified in the same order as in the definition of the derived type.

Example

```
   character(len=30) :: intfile
   integer :: ios,i=1,j=1,k=1
   real :: a=1.,b=1.,c=1.
   write (*,*) a,b,c       ! write values to stdout
                           ! using list directed i/o
   write(3,"(3i10)") i,j,k ! write to unit 3 using format
   write(10,*) i,j,k       ! write stdout using format
10 format(3i10)
   write(11) a,b,c ! write unformatted data to unit 11
   write(intfile,10) i,j,k       ! write internal file
   write(12, rec=2) a,b,c        ! write direct access file
   write(13,10,err=20) i,j       ! write with error branch
20 write(13,10,iostat=ios) a     ! write with status return
   write(13,10,advance='no') i,j ! next write on same line
```

A Fortran 77 Compatibility

This chapter discusses issues that affect the behavior of Fortran 77 and Fortran 90 code when processed by LF95.

Different Interpretation Under Fortran 95

Standard Fortran 95 is a superset of standard Fortran 90 and a standard-conforming Fortran 90 program will compile properly under Fortran 95. There are, however, two situations in which the program's interpretation may differ.

- The behavior of the SIGN intrinsic function is different if the second argument is negative real zero.
- Fortran 90 has more intrinsic procedures than Fortran 77. Therefore, a standard-conforming Fortran 77 program may have a different interpretation under Fortran 90 if it invokes a procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an EXTERNAL statement as recommended for non-intrinsic functions in the appendix to the Fortran 77 standard.

Different Interpretation Under Fortran 90

Standard Fortran 90 is a superset of standard Fortran 77 and a standard-conforming Fortran 77 program will compile properly under Fortran 90. There are, however, some situations in which the program's interpretation may differ.

- Fortran 77 permitted a processor to supply more precision derived from a REAL constant than can be contained in a REAL datum when the constant is used to initialize a DOUBLE PRECISION data object in a DATA statement. Fortran 90 does not permit this option.

- If a named variable that is not in a common block is initialized in a DATA statement and does not have the SAVE attribute specified, Fortran 77 left its SAVE attribute processor-dependent. Fortran 90 specifies that this named variable has the SAVE attribute.
- Fortran 77 required that the number of characters required by the input list must be less than or equal to the number of characters in the record during formatted input. Fortran 90 specifies that the input record is logically padded with blanks if there are not enough characters in the record, unless the PAD=”NO” option is specified in an appropriate OPEN statement.
- Fortran 90 has more intrinsic procedures than Fortran 77. Therefore, a standard-conforming Fortran 77 program may have a different interpretation under Fortran 90 if it invokes a procedure having the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an EXTERNAL statement as recommended for non-intrinsic functions in the appendix to the Fortran 77 standard.

Obsolescent Features

The following features are obsolescent or deleted from the Fortran 95 standard. While these features are still supported in LF95, their use in new code is not recommended:

- Arithmetic IF
- REAL and double-precision DO control variables and DO loop control expressions
- shared DO termination and termination on a statement other than END DO or CONTINUE
- Branching to an END IF statement from outside its IF block
- Alternate return
- PAUSE statement
- ASSIGN statement and assigned GOTO statement
- Assigned format specifier
- nH (Hollerith) edit descriptor
- Computed GOTO statement
- Statement functions
- DATA statements amongst executable statements
- Assumed-length CHARACTER functions
- Fixed-source form
- CHARACTER* form of CHARACTER declaration

B New in Fortran 95

The following Fortran 95 features were not present in Fortran 77. Fortran 95 features that were not present in Fortran 90 are marked with an asterisk.

Miscellaneous

free source form
enhancements to fixed source form:
    “;” statement separator
    “!” trailing comment
names may be up to 31 characters in length
both upper and lower case characters are accepted
INCLUDE line
relational operators in mathematical notation
enhanced END statement
IMPLICIT NONE
binary, octal, and hexadecimal constants
quotation marks around CHARACTER constants

Data

enhanced type declaration statements
new attributes:
    extended DIMENSION attribute
    ALLOCATABLE
    POINTER
    TARGET
    INTENT
    PUBLIC
    PRIVATE
kind and length type parameters
derived types
pointers

Operations

extended intrinsic operators
extended assignment
user-defined operators

Arrays

automatic arrays
allocatable arrays
assumed-shape arrays
array sections
array expressions
masked array assignment (WHERE statement and construct)
FORALL statement*

Execution Control

CASE construct
enhance DO construct
CYCLE statement
EXIT statement

Input/Output

binary, octal, and hexadecimal edit descriptors
engineering and scientific edit descriptors
namelist formatting
partial record capabilities (non-advancing I/O)
extra OPEN and INQUIRE specifiers

Procedures

keyword arguments
optional arguments
INTENT attribute
derived type actual arguments and functions
array-valued functions
recursive procedures
user-defined generic procedures
user-defined elemental procedures*
pure procedures*
specification of procedure interfaces
internal procedures

Modules

New Intrinsic Procedures

NULL*
PRESENT
numeric functions
- CEILING
- FLOOR
- MODULO

character functions
- ACHAR
- ADJUSTL
- ADJUSTR
- IACHAR
- LEN_TRIM
- REPEAT
- SCAN
- TRIM
- VERIFY

kind Functions
- KIND
- SELECTED_INT_KIND
- SELECTED_REAL_KIND

LOGICAL
numeric inquiry functions
- DIGITS
- EPSILON
- HUGE
- MAXEXPONENT
- MINEXPONENT
- PRECISION
- RADIX
- RANGE
- TINY

BIT_SIZE
bit manipulation functions
- BTEST
- IAND
- IBCLR
- IBITS
- IBSET
- IEOR
- IOR
- ISHFT

    - ISHFTC
    - NOT
- TRANSFER
- floating-point manipulation functions
    - EXPONENT
    - FRACTION
    - NEAREST
    - RRSPACING
    - SCALE
    - SET_EXPONENT
    - SPACING
- vector and matrix multiply functions
    - DOT_PRODUCT
    - MATMUL
- array reduction functions
    - ALL
    - ANY
    - COUNT
    - MAXVAL
    - MINVAL
    - PRODUCT
    - SUM
- array inquiry functions
    - ALLOCATED
    - LBOUND
    - SHAPE
    - SIZE
    - UBOUND
- array construction functions
    - MERGE
    - FSOURCE
    - PACK
    - SPREAD
    - UNPACK
- RESHAPE
- array manipulation functions
    - CSHFT
    - EOSHIFT
    - TRANSPOSE
- array location functions
    - MAXLOC
    - MINLOC
- ASSOCIATED

intrinsic subroutines

- CPU_TIME*
- DATE_AND_TIME
- MVBITS
- RANDOM_NUMBER
- RANDOM_SEED
- SYSTEM_CLOCK

C Intrinsic Procedures

The tables in this chapter offer a synopsis of procedures included with Lahey Fortran. For detailed information on individual procedures, see the chapter *“Alphabetical Reference”* on page 61.

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 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
ABS *CABS* *CDABS* *CQABS* *DABS* *QABS* *IABS* *I2ABS* *IIABS* *JIABS*	Numeric REAL_4 REAL_8 REAL_16 REAL_8 REAL_16 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	Numeric COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	Absolute Value.	Elemental
AIMAG *DIMAG* *QIMAG*	REAL REAL_8 REAL_16	COMPLEX COMPLEX_8 COMPLEX_16	Imaginary part of a complex number.	Elemental
AINT *DINT* *QINT*	REAL REAL_8 REAL_16	REAL REAL_8 REAL_16	Truncation to a whole number.	Elemental
ANINT *DNINT* *QNINT*	REAL REAL_8 REAL_16	REAL REAL_8 REAL_16	REAL representation of the nearest whole number.	Elemental
CEILING	INTEGER_4	REAL	Smallest INTEGER greater than or equal to a number.	Elemental
CMPLX *DCMPLX* *QCMPLX*	COMPLEX COMPLEX_8 COMPLEX_16	Numeric Numeric Numeric	Convert to type COMPLEX.	Elemental
CONJG *DCONJG* *QCONJG*	COMPLEX COMPLEX_8 COMPLEX_16	COMPLEX COMPLEX_8 COMPLEX_16	Conjugate of a complex number.	Elemental
DBLE *DREAL** *DFLOAT** *DBLEQ*	REAL_8 REAL_8 REAL_8 REAL_8	Numeric COMPLEX_8 INTEGER_4 REAL_16	Convert to double-precision REAL type.	Elemental

Table 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
DIM *DDIM* *QDIM* *IDIM* *I2DIM* *IIDIM* *JIDIM*	INTEGER or REAL REAL_8 REAL_16 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	INTEGER or REAL REAL_8 REAL_16 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	The difference between two numbers if the difference is positive; zero otherwise.	Elemental
DPROD	REAL_8	REAL_4	Double-precision REAL product.	Elemental
EXPO-NENT	REAL	REAL	Exponent part of the model representation of a number.	Elemental
FLOOR	INTEGER_4	REAL	Greatest INTEGER less than or equal to a number.	Elemental
FRAC-TION	REAL	REAL	Fraction part of the physical representation of a number.	Elemental
INT *IDINT** *IQINT** *IFIX** *INT2** *INT4** *HFIX** *IINT** *JINT** *IIDINT** *JIDINT** *IIFIX** *JIFIX**	INTEGER INTEGER INTEGER INTEGER INTEGER_2 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4 INTEGER_2 INTEGER_4 INTEGER_2 INTEGER_4	Numeric REAL_8 REAL_16 REAL_4 Numeric Numeric REAL_4 REAL_4 REAL_4 REAL_8 REAL_8 REAL_4 REAL_4	Convert to INTEGER type.	Elemental

Table 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
MAX	INTEGER or REAL	INTEGER or REAL	Maximum value.	Elemental
*AMAX0**	REAL_4	INTEGER_4		
*AMAX1**	REAL_4	REAL_4		
*DMAX1**	REAL_8	REAL_8		
*QMAX1**	REAL_16	REAL_16		
*MAX0**	INTEGER_4	INTEGER_4		
*MAX1**	INTEGER_4	REAL_4		
*I2MAX0**	INTEGER_2	INTEGER_2		
*IMAX0**	INTEGER_2	INTEGER_2		
*JMAX0**	INTEGER_4	INTEGER_4		
*IMAX1**	INTEGER_2	REAL_4		
*JMAX1**	INTEGER_4	REAL_4		
*AIMAX0**	REAL_4	INTEGER_2		
*AJMAX0**	REAL_4	INTEGER_4		
MIN	INTEGER or REAL	INTEGER or REAL	Minimum value.	Elemental
*AMIN0**	REAL_4	INTEGER_4		
*AMIN1**	REAL_4	REAL_4		
*DMIN1**	REAL_8	REAL_8		
*QMIN1**	REAL_16	REAL_16		
*MIN0**	INTEGER_4	INTEGER_4		
*MIN1**	INTEGER_4	REAL_4		
*I2MIN0**	INTEGER_2	INTEGER_2		
*IMIN0**	INTEGER_2	INTEGER_2		
*JMIN0**	INTEGER_4	INTEGER_4		
*IMIN1**	INTEGER_2	REAL_4		
*JMIN1**	INTEGER_4	REAL_4		
*AIMIN0**	REAL_4	INTEGER_2		
*AJMIN0**	REAL_4	INTEGER_4		

Table 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
MOD *AMOD* *DMOD* *QMOD* *I2MOD* *IMOD* *JMOD*	INTEGER or REAL REAL_4 REAL_8 REAL_16 INTEGER_2 INTEGER_2 INTEGER_4	INTEGER or REAL REAL_4 REAL_8 REAL_16 INTEGER_2 INTEGER_2 INTEGER_4	Remainder.	Elemental
MODULO	INTEGER or REAL	INTEGER or REAL	Modulo.	Elemental
NEAREST	REAL	REAL	Nearest number of a given data type in a given direction.	Elemental
NINT *IDNINT* *IQNINT* *I2NINT* *ININT* *JNINT* *IIDNNT* *JIDNNT*	INTEGER INTEGER_4 INTEGER_4 INTEGER_2 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	REAL REAL_8 REAL_16 REAL REAL_4 REAL_4 REAL_8 REAL_8	Nearest INTEGER.	Elemental
REAL *FLOAT** *SNGL** *SNGLQ** *FLOATI** *FLOATJ** *DFLOTI** *DFLOTJ**	REAL REAL_4 REAL_4 REAL_4 REAL_4 REAL_4 REAL_8 REAL_8	Numeric INTEGER REAL_8 REAL_16 INTEGER_2 INTEGER_4 INTEGER_2 INTEGER_4	Convert to REAL type.	Elemental
RRSPACING	REAL	REAL	Reciprocal of relative spacing near a given number.	Elemental

Table 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
SCALE	REAL	REAL and INTEGER	Multiply a number by a power of two.	Elemental
SET_ EXPO-NENT	REAL	REAL and INTEGER	Model representation of a number with exponent part set to a power of two.	Elemental
SIGN *DSIGN* *QSIGN* *ISIGN* *I2SIGN* *IISIGN* *JISIGN*	INTEGER or REAL REAL_8 REAL_16 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	INTEGER or REAL REAL_8 REAL_16 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	Transfer of sign.	Elemental
SPACING	REAL	REAL	Absolute spacing near a given number.	Elemental

Table 13: Mathematical Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
ACOS *DACOS*	REAL REAL_8	REAL REAL_8	Arccosine.	Elemental
ASIN *DASIN*	REAL REAL_8	REAL REAL_8	Arcsine.	Elemental
ATAN *DATAN*	REAL REAL_8	REAL REAL_8	Arctangent.	Elemental
ATAN2 *DATAN2*	REAL REAL_8	REAL REAL_8	Arctangent of y/x (principal value of the argument of the complex number (x,y)).	Elemental
COS *CCOS* *CDCOS* *CQCOS* *DCOS* *QCOS*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	Cosine.	Elemental
COSH *DCOSH* *QCOSH*	REAL REAL_8 REAL_16	REAL REAL_8 REAL_16	Hyperbolic cosine.	Elemental
EXP *CEXP* *CDEXP* *CQEXP* *DEXP* *QEXP*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	Exponential.	Elemental

Table 13: Mathematical Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
LOG *ALOG* *CLOG* *CDLOG* *CQLOG* *DLOG* *QLOG*	REAL or COMPLEX REAL_4 COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	REAL or COMPLEX REAL_4 COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	Natural logarithm.	Elemental
LOG10 *ALOG10* *DLOG10* *QLOG10*	REAL REAL_4 REAL_8 REAL_16	REAL REAL_4 REAL_8 REAL_16	Common logarithm.	Elemental
SIN *CSIN* *CDSIN* *CQSIN* *DSIN* *QSIN*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	Sine.	Elemental
SINH *DSINH* *QSINH*	REAL REAL_8 REAL_16	REAL REAL_8 REAL_16	Hyperbolic sine.	Elemental
SQRT *CSQRT* *CDSQRT* *CQSQRT* *DSQRT* *QSQRT*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	REAL or COMPLEX COMPLEX_4 COMPLEX_8 COMPLEX_16 REAL_8 REAL_16	Square root.	Elemental
TAN *DTAN* *QTAN*	REAL REAL_8 REAL_16	REAL REAL_8 REAL_16	Tangent.	Elemental

Table 13: Mathematical Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
TANH *DTANH* *QTANH*	REAL REAL_8 REAL_16	REAL REAL_8 REAL_16	Hyperbolic tan-gent.	Elemental

Table 14: Character Functions

Name	Description	Class
ACHAR	Character in a specified position of the ASCII collating sequence.	Elemental
ADJUSTL	Adjust to the left, removing leading blanks and inserting trailing blanks.	Elemental
ADJUSTR	Adjust to the right, removing trailing blanks and inserting leading blanks.	Elemental
CHAR	Given character in the collating sequence of the a given character set.	Elemental
IACHAR	Position of a character in the ASCII collating sequence.	Elemental
ICHAR	Position of a character in the processor collating sequence associated with the kind of the character.	Elemental
INDEX	Starting position of a substring within a string.	Elemental
LEN	Length of a CHARACTER data object.	Inquiry
LEN_TRIM	Length of a CHARACTER entity without trailing blanks.	Elemental
LGE	Test whether a string is lexically greater than or equal to another string based on the ASCII collating sequence.	Elemental
LGT	Test whether a string is lexically greater than another string based on the ASCII collating sequence.	Elemental
LLE	Test whether a string is lexically less than or equal to another string based on the ASCII collating sequence.	Elemental
LLT	Test whether a string is lexically less than another string based on the ASCII collating sequence.	Elemental
REPEAT	Concatenate copies of a string.	Transformational
SCAN	Scan a string for any one of a set of characters.	Elemental

Table 14: Character Functions

Name	Description	Class
TRIM	Omit trailing blanks.	Transforma-tional
VERIFY	Verify that a set of characters contains all the characters in a string.	Elemental

Table 15: Array Functions

Name	Description	Class
ALL	Determine whether all values in a mask are true along a given dimension.	Transformational
ALLOCATED	Indicate whether an allocatable array has been allocated.	Inquiry
ANY	Determine whether any values are true in a mask along a given dimension.	Transformational
COUNT	Count the number of true elements in a mask along a given dimension.	Transformational
CSHIFT	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.	Transformational
DOT_ PRODUCT	Dot-product multiplication of vectors.	Transformational
EOSHIFT	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.	Transformational
LBOUND	Lower bounds of an array or a dimension of an array.	Inquiry
MATMUL	Matrix multiplication.	Transformational
MAXLOC	Location of the first element in *array* having the maximum value of the elements identified by *mask.*	Transformational
MAXVAL	Maximum value of elements of an array, along a given dimension, for which a mask is true.	Transformational
MERGE	Choose alternative values based on the value of a mask.	Elemental

Table 15: Array Functions

Name	Description	Class
MINLOC	Location of the first element in *array* having the minimum value of the elements identified by *mask*.	Transforma-tional
MINVAL	Minimum value of elements of an array, along a given dimension, for which a mask is true.	Transforma-tional
PACK	Pack an array into a vector under control of a mask.	Transforma-tional
PRODUCT	Product of elements of an array, along a given dimension, for which a mask is true.	Transforma-tional
RESHAPE	Construct an array of a specified shape from a given array.	Transforma-tional
SHAPE	Shape of an array.	Inquiry
SIZE	Size of an array or a dimension of an array.	Inquiry
SPREAD	Adds a dimension to an array by adding copies of a data object along a given dimension.	Transforma-tional
SUM	Sum of elements of an array, along a given dimension, for which a mask is true.	Transforma-tional
TRANSPOSE	Transpose an array of rank two.	Transforma-tional
UBOUND	Upper bounds of an array or a dimension of an array.	Inquiry
UNPACK	Unpack an array of rank one into an array under control of a mask.	Transforma-tional

Table 16: Inquiry and Kind Functions

Name	Description	Class
ALLOCATED	Indicate whether an allocatable array has been allocated.	Inquiry
ASSOCIATED	Indicate whether a pointer is associated with a target.	Inquiry
BIT_SIZE	Size, in bits, of a data object of type INTEGER.	Inquiry
DIGITS	Number of significant binary digits.	Inquiry
EPSILON	Positive value that is almost negligible compared to unity.	Inquiry
HUGE	Largest representable number of data type.	Inquiry
KIND	Kind type parameter.	Inquiry
LBOUND	Lower bounds of an array or a dimension of an array.	Inquiry
LEN	Length of a CHARACTER data object.	Inquiry
MAXEXPONENT	Maximum binary exponent of data type.	Inquiry
MINEXPONENT	Minimum binary exponent of data type.	Inquiry
PRECISION	Decimal precision of data type.	Inquiry
PRESENT	Determine whether an optional argument is present.	Inquiry
RADIX	Number base of the physical representation of a number.	Inquiry
RANGE	Decimal range of the data type of a number.	Inquiry
SELECTED_INT_KIND	Kind type parameter of an INTEGER data type that represents all integer values *n* with $-10^r < n < 10^r$.	Transformational
SELECTED_REAL_KIND	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*.	Transformational

Table 16: Inquiry and Kind Functions

Name	Description	Class
SHAPE	Shape of an array.	Inquiry
SIZE	Size of an array or a dimension of an array.	Inquiry
TINY	Smallest representable positive number of data type.	Inquiry
UBOUND	Upper bounds of an array or a dimension of an array.	Inquiry

Table 17: Bit Manipulation Procedures

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
BTEST *BITEST* *BJTEST*	LOGICAL_4 LOGICAL_4 LOGICAL_4	INTEGER INTEGER_2 INTEGER_4	Bit testing.	Elemental
IAND *IIAND* *JIAND*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Bit-wise logical AND.	Elemental
IBCLR *IIBCLR* *JIBCLR*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Clear one bit to zero.	Elemental
IBITS *IIBITS* *JIBITS*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Extract a sequence of bits.	Elemental
IBSET *IIBSET* *JIBSET*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Set a bit to one.	Elemental
IEOR *IIEOR* *JIEOR*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Bit-wise logical exclusive OR.	Elemental
IOR *IIOR* *JIOR*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Bit-wise logical inclusive OR.	Elemental
ISHFT *IISHFT* *JISHFT*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Bit-wise shift.	Elemental
ISHFTC *IISHFTC* *JISHFTC*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Bit-wise circular shift of rightmost bits.	Elemental
MVBITS		INTEGER	Copy a sequence of bits from one INTEGER data object to another.	Subroutine

Table 17: Bit Manipulation Procedures

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
NOT *INOT* *JNOT*	INTEGER INTEGER_2 INTEGER_4	INTEGER INTEGER_2 INTEGER_4	Bit-wise logical complement.	Elemental

Table 18: Other Intrinsic Functions

Name	**Description**	**Class**
LOGICAL	Convert between kinds of LOGICAL.	Elemental
NULL	Disassociated pointer.	Elemental
TRANSFER	Interpret the physical representation of a number with the type and type parameters of a given number.	Transformational

Table 19: Standard Intrinsic Subroutines

Name	**Description**	**Class**
CPU_TIME	CPU time.	Subroutine
DATE_AND_TIME	Date and real-time clock data.	Subroutine
MVBITS	Copy a sequence of bits from one INTEGER data object to another.	Subroutine
RANDOM_NUMBER	Uniformly distributed pseudorandom number or numbers in the range $0 \le x < 1$.	Subroutine
RANDOM_SEED	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.	Subroutine
SYSTEM_CLOCK	INTEGER data from the real-time clock.	Subroutine

Table 20: VAX/IBM Intrinsic Functions Without Fortran 90 Equivalents

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
ACOSD *DACOSD* *QACOSD*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Arccosine in degrees.	Elemental
ALGAMA *DLGAMA* *QLGAMA*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Log gamma function.	Elemental
ASIND *DASIND* *QASIND*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Arcsine in degrees.	Elemental
ATAND *DATAND* *QATAND*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Arctangent in degrees.	Elemental
ATAN2D *DATAN2D* *QATAN2D*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Arctangent of *y*/*x* (principal value of the argument of the complex number (*x*,*y*)) in degrees.	Elemental
COSD *DCOSD* *QCOSD*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Cosine in degrees.	Elemental
COTAN *DCOTAN* *QCOTAN*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Cotangent.	Elemental
ERF *DERF* *QERF*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Error function.	Elemental
ERFC *DERFC* *QERFC*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Error function complement.	Elemental

Table 20: VAX/IBM Intrinsic Functions Without Fortran 90 Equivalents

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
GAMMA *DGAMMA* *QGAMMA*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Gamma function.	Elemental
SIND *DSIND* *QSIND*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Sine in degrees.	Elemental
TAND *DTAND* *QTAND*	REAL_4 REAL_8 REAL_16	REAL_4 REAL_8 REAL_16	Tangent in degrees.	Elemental
IZEXT *IZEXT2* *JZEXT* *JZEXT2* *JZEXT4*	INTEGER_2 INTEGER_2 INTEGER_4 INTEGER_4 INTEGER_4	LOGICAL_1 INTEGER_2 LOGICAL_4 INTEGER_2 INTEGER_4	Zero extend.	Elemental

Table 21: Utility Procedures

Name	Description	Class
CARG	Pass *item* to a procedure as a C data type by value. CARG can only be used as an actual argument.	Utility Function
DLL_EXPORT	Specify which procedures should be available in a dynamic-link library.	Utility Subroutine
DLL_IMPORT	Specify which procedures are to be imported from a dynamic-link library.	Utility Subroutine
DVCHK	The initial invocation of the DVCHK subroutine masks the divide-by-zero interrupt on the floating-point unit. Subsequent invocations return true or false in the *lflag* variable if the exception has occurred or not occurred, respectively. DVCHK will not check or mask zero divided by zero. Use INVALOP to check for a zero divided by zero.	Utility Subroutine
ERROR	Print a message to the console with a subprogram traceback, then continue processing.	Utility Subroutine
EXIT	Terminate the program and set the DOS error level.	Utility Subroutine
FLUSH	Empty the buffer for an input/output unit by writing to its corresponding file. Note that this does not flush the DOS file buffer.	Utility Subroutine
GETCL	Get command line.	Utility Subroutine
GETENV	Get the specified environment variable.	Utility Function
INVALOP	The initial invocation of the INVALOP subroutine masks the invalid operator interrupt on the floating-point unit. Subsequent invocations return true or false in the *lflag* variable if the exception has occurred or not occurred, respectively.	Utility Subroutine
IOSTAT_MSG	Get a runtime I/O error message then continue processing.	Utility Subroutine
NDPERR	Report floating point exceptions.	Utility Function

Table 21: Utility Procedures

Name	Description	Class
NDPEXC	Mask all floating point exceptions.	Utility Subroutine
OFFSET	Get the DOS offset portion of the memory address of a variable, substring, array reference, or external subprogram.	Utility Function
OVEFL	The initial invocation of the OVEFL subroutine masks the overflow interrupt on the floating-point unit. Subsequent invocations return true or false in the *lflag* variable if the exception has occurred or not occurred, respectively.	Utility Subroutine
POINTER	Get the memory address of a variable, substring, array reference, or external subprogram.	Utility Function
PRECFILL	Set fill character for numeric fields that are wider than supplied numeric precision. The default is ’0’.	Utility Subroutine
PROMPT	Set prompt for subsequent READ statements. Fortran default is no prompt.	Utility Subroutine
SEGMENT	Get the DOS segment portion of the memory address of a variable, substring, array reference, or external subprogram.	Utility Function
SYSTEM	Execute a DOS command as if from the DOS command line.	Utility Subroutine
UNDFL	The initial invocation of the UNDFL subroutine masks the underflow interrupt on the floating-point unit. Subsequent invocations return true or false in the *lflag* variable if the exception has occurred or not occurred, respectively.	Utility Subroutine
VAL	Pass an item to a procedure by value. VAL can only be used as an actual argument.	Utility Function
YIELD	Causes a Windows 3.1 program to yield control to Windows so that computation-intensive operations do not monopolize the processor. YIELD has no effect under other supported operating systems.	Utility Function

D Porting Extensions

The following non-standard features are supported by LF95. Note that for service procedures, a module SERVICE_ROUTINES is provided. Use SERVICE_ROUTINES to have the compiler check interfaces for the various service procedures. See the USE statement for details on how to use a module.

- Dollar sign as a letter
- Backslash as a special character
- Unlimited number of continuation lines in free or fixed source form
- Omission of required significant blanks in free source form
- DO UNTIL statement
- FIND statement
- STRUCTURE statement
- END STRUCTURE statement
- UNION statement
- END UNION statement
- MAP statement
- END MAP statement
- RECORD statement
- Non-standard POINTER statement
- AUTOMATIC statement and attribute
- STATIC statement and attribute
- VALUE statement and attribute
- VOLATILE statement and attribute

- DLL_IMPORT statement
- DLL_EXPORT statement
- BYTE statement
- Double-precision COMPLEX constants
- Hollerith constants
- B*digits* form of binary constant
- *digits*O form of octal constant
- X'*digits*' form of hexadecimal constant
- '*digits*'X form of hexadecimal constant
- Zdigits form of hexadecimal constant
- Binary, Octal, or Hexadecimal constant in a DATA, PARAMETER, or type declaration statement
- '.' period structure component separator
- *type*n* form in type declaration, FUNCTION or IMPLICIT statement (e.g. INTEGER*4)
- /*literal-constant*/ form of initialization in type declaration statement
- IMPLICIT UNDEFINED statement
- Namelist input/output on internal file
- Variable format expressions
- NUM specifier
- ACTION = 'BOTH'
- FORM = 'TRANSPARENT' (use FORM=BINARY instead)
- TOTALREC specifier
- STATUS = 'SHR'
- G*w* edit descriptor
- $ edit descriptor
- \ edit descriptor
- R edit descriptor
- D, E, F, G, I, L, B, O or Z descriptor without *w*, *d* or *e* indicators
- &*name*...&end namelist record

- TIMER intrinsic subroutine
- SEGMENT and OFFSET intrinsic functions
- VAL and LOC intrinsic functions
- The following service subroutines: ABORT, BEEP, BIC, BIS, CLOCK, CLOCKM, DATE, ERRSAV, ERRSTR, ERRSET, ERRTRA, FDATE, FLUSH, FREE, GETARG, GETDAT, GETLOG, GETPARM, GETTIM, GMTIME, IBTOD, IDATE, IETOM, ITIME, IVALUE, LTIME, MTOIE, PERROR, PRECFILL, PRNSET, QSORT, SETRCD, SETBIT, SLEEP, TIME
- The following service functions: ACCESS, ALARM, BIT, CHDIR, CHMOD, CTIME, DRAND, DTIME, ETIME, FGETC, FPUTC, FSEEK, FSTAT, FTELL, GETC, GETCWD, GETFD, GETPID, HOSTNM, IARGC, IERRNO, INMAX, IRAND, JDATE, KILL, LNBLNK, LONG, LSTAT, MALLOC, NARGS, PUTC, RAN, RAND, RENAME, RINDEX, RTC, SECOND, SECNDS, SETDAT, SETTIM, SHORT, SIGNAL, STAT, TIMEF, UNLINK

Additional information on service routines is in the file `readme_service_routines.txt` (Windows) or `service_routines` (Linux).

E Glossary

action statement: A single statement specifying a computational action.

actual argument: An expression, a variable, a procedure, or an alternate return specifier that is specified in a procedure reference.

allocatable array: A named array having the ALLOCATABLE attribute. Only when it has space allocated for it does it have a shape and may it be referenced or defined.

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 ahead of 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*’’.

association: Name association, pointer association, or storage association.

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 assigned to a data object that modifies how the object behaves. The property may be specified in a type declaration statement, or in an attribute assignment 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.

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 must lie.

character: A letter, digit, or other symbol.

character string: A sequence of characters numbered from left to right 1, 2, 3, . . .

collating sequence: The order of all the different characters in a particular character set.

common block: A block of physical storage that may be accessed by any of the scoping units in an executable program.

component: A constituent of a derived type.

conformable: Two arrays are said to be conformable if they have the same shape. A scalar is conformable with any array.

conformance: An executable program conforms to the standard if it uses only those forms and relationships described therein and if the executable program has an interpretation according to the standard. A program unit conforms to the standard if it can be included in an executable program in a manner that allows the executable program to be standard conforming. A processor conforms to the standard if it executes standard-conforming programs in a manner that fulfills the interpretations prescribed in the standard.

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 must not change during execution of an executable 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 a CASE, DO, IF, or WHERE statement and ending with the corresponding terminal statement.

contiguous: Having the property of being adjoining, or adjacent to.

data: A set of quantities that may have any of the set of values specified for its data type.

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 or variable.

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 single quantity that may have any of the set of values specified for its data type.

deferred shape: The declaration of an arrays rank only, leaving the size and shape of the array undefined.

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` 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.

derived type: A type whose data have components, each of which is either of intrinsic type or of another derived type.

designator: See subobject designator.

disassociated: 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 pointer: A dummy argument that is a pointer.

dummy procedure: A dummy argument that is specified or referenced as a procedure.

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 operator, an interface block, a common block, an external unit, a statement function, a type, a named variable, an expression, a component of a structure, a named constant, a statement label, a construct, or a namelist group.

executable construct: A CASE, DO, IF, or WHERE construct or an action statement.

executable program: A set of program units that includes exactly one main program.

executable statement: An instruction to perform or control one or more computational actions.

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.

extent: The size of one dimension of an array.

external file: A sequence of records that exists in a medium external to the executable 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 contained 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 non-negative INTEGER.

file: An internal file or an external file.

function: A procedure that is invoked in an 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.

global entity: An entity identified by a lexical token whose scope is an executable program. It may be a program unit, a common block, or an external procedure.

host: A main program or subprogram that contains an internal procedure is called the host of the internal procedure. A module that contains a module procedure is called the host of the module procedure.

host association: The process by which an internal subprogram, module subprogram, or derived type definition accesses entities of its host.

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.

initialization expression: An expression that can be evaluated at compile time.

inquiry function: An intrinsic function whose result depends on properties of the principal argument other than the value of the argument.

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 contained in a main program or another subprogram.

intrinsic: An adjective applied to types, operations, assignment statements, and procedures 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.

keyword argument: The association of a calling program's argument with the subprogram's dummy argument by assigning a value to the dummy argument's keyword. Keywords are associated with dummy arguments using either an implicit or explicit interface.

kind type parameter: A parameter whose values label the available kinds of an intrinsic type.

label: See statement label.

length of a character string: The number of characters in the character string.

lexical token: A sequence of one or more characters with an indivisible interpretation.

line: A source-form record containing from 0 to 132 characters.

literal constant: A constant without a name.

local entity: An entity identified by a lexical token whose scope is a scoping unit.

logical mask: An array of logical values which can be derived from a logical array variable or a logical array expression.

main program: A program unit that is not a module, subprogram, or block data program unit.

model representation: A formula which describes the finite set of numbers representable by a digital computer.

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 contained in a module but is not an internal subprogram.

name: A lexical token consisting of a letter followed by up to 30 alphanumeric characters (letters, digits, and underscores).

name association: Argument association, use association, or host association.

named: Having a name.

named constant: A constant that has a name.

numeric type: INTEGER, REAL or COMPLEX type.

object: Data object.

obsolescent feature: A feature that was considered to have been redundant in FORTRAN 77 but that is still in frequent use. Obsolescent features have modern counterparts that allow a greater measure of safety with less effort on the part of the programmer.

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.

pointer: A variable that has the POINTER attribute. A pointer must 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.

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-name => 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.

positional argument: The association of a calling program’s argument list with the subprogram’s dummy argument list in sequential order.

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 procedure or is not a dummy argument of the invoking procedure.

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 interface: The characteristics of a procedure, the name of the procedure, the name of each dummy argument, and the generic identifiers (if any) by which it may be referenced.

processor: The combination of a computing system and the mechanism by which executable programs are transformed for use on that computing system.

program: See executable program and main program.

program unit: The fundamental component of an executable program. A sequence of statements and comment lines. It may be a main program, a module, an external subprogram, or a block data program unit.

rank: The number of dimensions of an array. Zero for a scalar.

record: A sequence of values 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, or 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.

scalar:

A single datum that is not an array.

Not having the property of being an array.

scope: That part of an executable program within which a lexical token has a single interpretation. It may be an executable program, a scoping unit, 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 contained within it, or

A program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms contained within 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.

shape: The rank and extents of an array. The shape of an array may be represented by a rank-one array whose size is the rank of the array, and whose elements are the extents of each dimension.

size: For an array, the total number of elements.

specification expression: A scalar INTEGER expression that can be evaluated on entry to the program unit at the time of execution.

statement: A sequence of lexical tokens. It usually consists of a single line, but the ampersand symbol may be used to continue a statement 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 keyword: A word that is part of the syntax of a statement and that may be used to identify the 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.

stride: The increment specified in a subscript triplet.

string delimiter: A character which is used in source code to mark the beginning and end of character data. Fortran string delimiters are the apostrophe (‘) and the quote (“).

structure: A scalar data object of derived type.

structure component: The part of a data object of derived type corresponding to a component of its type.

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, or a substring.

subobject designator: A name, followed by one or more of the following: component selectors, array section selectors, array element selectors, and substring selectors.

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 list 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 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.

type: Data type.

type declaration statement: An INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, LOGICAL, or TYPE statement.

type parameter: A parameter of an intrinsic data type. KIND= and LEN= are the type parameters.

type parameter values: The values of the type parameters of a data entity of an intrinsic data type.

ultimate component: For a derived-type or a structure, a component that is of intrinsic type or has the POINTER attribute, or an ultimate component of a component that is a derived type and does not have the POINTER attribute.

undefined: For a data object, the property of not having a determinate value.

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 an executable 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 without a subscript reference.

F 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.

DOS uses `<control-Z>` (`^Z`) for the end-of-file delimiter and `<control-M>` (`^M`) for carriage return. To enter these two characters in a CHARACTER constant, use concatenation and the CHAR function.

Attempting to input or output `^Z` (end-of-file), `^M` (new line), or `^C` (break) in a sequential file is not recommended and may produce undesirable results.

Table 22: 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

Table 22: ASCII Chart

Character	HEX Value	Decimal Value	ASCII Abbr.	Description
^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
#	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

Table 22: ASCII Chart

Character	HEX Value	Decimal Value	ASCII Abbr.	Description
.	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	=	equals
>	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

Table 22: ASCII Chart

Character	HEX Value	Decimal Value	ASCII Abbr.	Description
F	46	70	F	uppercase F
G	47	71	G	uppercase G
H	48	72	H	uppercase H
I	49	73	I	uppercase I
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

Table 22: ASCII Chart

Character	HEX Value	Decimal Value	ASCII Abbr.	Description
^	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
q	71	113	LCQ	lowercase q
r	72	114	LCR	lowercase r
s	73	115	LCS	lowercase s
t	74	116	LCT	lowercase t

Table 22: ASCII Chart

Character	HEX Value	Decimal Value	ASCII Abbr.	Description
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

Index