Lahey Fortran 90 Language Reference

Revision D

Lahey Computer Systems, Inc.

P. O. Box 6091
Incline Village, NV 89450

Recycled Paper

Printed on 50%
recycled paper

Copyright

Trademarks

Disclaimer

Lahey Computer Systems, Inc.
865 Tahoe Boulevard
P.O. Box 6091
Incline Village, NV 89450-6091
(702) 831-2500
Fax: (702) 831-8123

http://www.lahey.com

Technical Support
(702) 831-2500
support@lahey.com

Table of Contents

Introduction

Lahey Fortran 90 is a complete implementation of the ANSI and ISO Fortran 90 standards. Numerous popular extensions are supported.

This manual is intended as a reference to the Fortran 90 language for programmers with experience in Fortran. For information on creating programs using the Lahey Fortran 90 Language System, see the Lahey Fortran 90 User's Guide.

Manual Organization

The manual is organized in six 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 Lahey Fortran 90.
- Appendix B, *New in Fortran 90*, lists Fortran 90 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 Lahey Fortran 90.
- Appendix D, *Glossary*, defines various technical terms used in this manual.
- Appendix E, *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 90 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, can be up to 31 characters in length and consists entirely of letters, digits, and underscores. In fixed source form, a name can contain blanks, which are ignored.

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_tooooooooooooooooooooooooooooooo_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 Lahey Fortran, 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 systems. The kinds available in Lahey Fortran are summarized in the following table:

Table 1: Intrinsic Data Types

Type	Kind Type Parameter	Notes
INTEGER	1	Range: -127 to 127
INTEGER	2	Range: -32,767 to 32,767
INTEGER	4*	Range: -2,147,483,647 to 2,147,483,647
REAL	4*	Range: $1.18 * 10^{-38}$ to $3.40 * 10^{38}$ Precision: 7-8 decimal digits
REAL	8	Range: $2.23 * 10^{-308}$ to $1.79 * 10^{308}$ Precision: 15-16 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
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 Lahey Fortran). 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 Lahey Fortran).

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 and `D` for double 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 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 Elf90). 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 15).

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 106).
- PARAMETER — the data object is a named constant (see *"PARAMETER Statement"* on page 186).
- 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 188).
- TARGET — the data object that is to be aliased by a POINTER data object (see *"TARGET Statement"* on page 222).
- EXTERNAL — the name is that of an external procedure (see *"EXTERNAL Statement"* on page 126).
- 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 63).
- INTENT — the dummy argument value will not change in a procedure (INTENT (IN)), will not be provided an initial value by the calling subprogram (INTENT (OUT)), or both an initial value will be provided and a new value may result (INTENT (IN OUT)) (see *"INTENT Statement"* on page 150).
- 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 195).
- PRIVATE — the named data object or procedure in a MODULE program unit is accessible only in the current module (see *"PRIVATE Statement"* on page 193).

- INTRINSIC — the name is that of an intrinsic function (see *“INTRINSIC Statement”* on page 153).
- 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 184).
- SAVE — the data object retains its value, association status, and allocation status after a RETURN or END statement (see *“SAVE Statement”* on page 207).
- 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 211).

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 11).

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 ith dimension.

j_i is the lower bound of the ith dimension.

d_i is the size of the ith dimension.

n is the rank of the array.

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 12 and *"Array Pointers"* on page 12). 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 13).
- A dummy array can be of undeclared size (*"Assumed-Size Arrays"* on page 13).
- An array can have variable dimensions based on the values of dummy arguments (*"Adjustable and Automatic Arrays"* on page 14).

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 49).

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 108).

```
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 initialized in the derived type definition and cannot be a function. 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 54) 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 51).

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. Units 5 and 6 can be connected to other physical devices or files. 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 181).

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 - record zero is a header
- stored as CHARACTER data
- disk files only
- records can be accessed in any order
- not easily processed outside of Lahey Fortran
- 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 Lahey Fortran

Unformatted Direct

- fixed-length records - record zero is a header
- stored as binary data
- disk files only
- records can be accessed in any order
- fastest
- not easily read outside of Lahey Fortran

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 Lahey Fortran
- 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 Charac-ter	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 30). 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 128 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, 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 filledwith 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*. The output width is padded with blanks if the number is smaller than the field. Note that an input width must always be specified.

REAL Editing (F, D, and E)

The 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.

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.
- E or D followed by an optionally signed INTEGER constant.

For 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.

For 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. an 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 E and D editing, the scale factor *k* controls the position of the decimal point. If $-d < k \le 0$, 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 29) 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 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 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.

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.

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.

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.

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

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE,
or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
statements</td><td>IMPLICIT
statements</td></tr>
<tr><td>PARAMETER
and DATA
statements</td><td>Derived-type definitions,
interface blocks,
type declaration statements,
statement function statements,
and specification statements</td></tr>
<tr><td>DATA statements</td><td>Executable statements</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal subprograms or module subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Statements are restricted in what *scoping units* (see *"Scope"* on page 56) 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 81 and *“IF Construct”* on page 138).
- The DO construct controls how many times a block will be executed (see *“DO Construct”* on page 108).
- The WHERE construct controls which elements of an array will be affected by a block of assignment statements (see *“WHERE Construct”* on page 233).

Construct Names

Optional construct names can be used with CASE, IF, and DO 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

<table>
<tr><td rowspan="6">Functions</td><td rowspan="2">Intrinsic Functions</td><td>Generic Intrinsic Functions</td></tr>
<tr><td>Specific Intrinsic Functions</td></tr>
<tr><td rowspan="2">External Functions</td><td>Generic External Functions</td></tr>
<tr><td>Specific External Functions</td></tr>
<tr><td colspan="2">Internal Functions</td></tr>
<tr><td colspan="2">Statement Functions</td></tr>
<tr><td rowspan="5">Subroutines</td><td rowspan="2">Intrinsic Subroutines</td><td>Generic Intrinsic Subroutines</td></tr>
<tr><td>Specific Intrinsic Subroutines</td></tr>
<tr><td rowspan="2">External Sub-routines</td><td>Generic External Subroutines</td></tr>
<tr><td>Specific External Subroutines</td></tr>
<tr><td colspan="2">Internal Subroutines</td></tr>
</table>

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, intrinsic procedures can be *elemental* or *non-elemental*. An elemental intrinsic 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 249.

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 Statement"* on page 217) 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 43:

```
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 131).

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 49).

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 49). 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 labelled 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 Statement"* on page 151) 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 Statement"* on page 151).

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 Statement"* on page 151). The mechanism is similar to that used to resolve a defined binary operation (see *"Defined Operations"* on page 51), 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 43 and *"Intrinsic Procedures"* on page 42.

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).

In Lahey Fortran, any module program units must appear before any other program units in a source file.

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 Lahey Fortran. 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.

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 89), or declare it in a module and use the module (see *“Module Program Units”* on page 54).

2 Alphabetical Reference

ABS Function

Description

Absolute value.

Syntax

ABS (a)

Arguments

a must be of type REAL, INTEGER, or COMPLEX.

Result

If *a* is of type INTEGER or REAL, the result is of the same type as *a* and has the value $|a|$; if *a* is COMPLEX with value (x,y), the result is a REAL representation of $\sqrt{x^2 + y^2}$.

Example

```
x = abs(-4.2)  ! x is assigned the value 4.2
```

ACHAR Function

Description

Character in a specified position of the ASCII collating sequence.

Syntax

ACHAR (*i*)

Arguments

i must be of type INTEGER.

Result

A CHARACTER of length one that is the character in position (*i*) of the ASCII collating sequence.

Example

```
c = achar(65)  ! c is assigned the value 'A'
```

ACOS Function

Description

Arccosine.

Syntax

ACOS (*x*)

Arguments

x must be of type REAL and must be within the range $-1 \leq x \leq 1$.

Result

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

Example

```
r = acos(.5)  ! r is assigned the value 1.04720
```

ADJUSTL Function

Description

Adjust to the left, removing leading blanks and inserting trailing blanks.

Syntax
ADJUSTL (*string*)

Arguments
string must be of type CHARACTER.

Result
A CHARACTER of the same length and kind as *string*. Its value is the same as that of *string* except that any leading blanks have been deleted and the same number of trailing blanks has been inserted.

Example
```
adjusted = adjustl('   string')
         ! adjusted is assigned the value 'string   '
```

ADJUSTR Function

Description
Adjust to the right, removing trailing blanks and inserting leading blanks.

Syntax
ADJUSTR (*string*)

Arguments
string must be of type CHARACTER.

Result
A CHARACTER of the same length and kind as *string*. Its value is the same as that of *string* except that any trailing blanks have been deleted and the same number of leading blanks has been inserted.

Example
```
adjusted = adjustr('string   ')
         ! adjusted is assigned the value '   string'
```

AIMAG Function

Description
Imaginary part of a complex number.

Syntax

AIMAG (*z*)

Arguments

z must be of type COMPLEX.

Result

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

Example

```
r = aimag(-4.2,5.1)  ! r is assigned the value 5.1
```

AINT Function

Description

Truncation to a whole number.

Syntax

AINT (*a*, *kind*)

Required Arguments

a must be of type REAL.

Optional Arguments

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

Result

A REAL value with the kind specified by *kind*, if present; otherwise with the kind of *a*. The result is equal to the value of *a* without its fractional part.

Example

```
r = aint(-7.32,2)  ! r is assigned the value -7.0
                   ! with kind 2
```

ALL Function

Description

Determine whether all values in a mask are true along a given dimension.

Syntax

ALL (*mask*, *dim*)

Required Arguments

mask must be of type LOGICAL. It must not be scalar.

Optional Arguments

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

Result

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

1. If *dim* is absent or *mask* has rank one, the result is scalar. The result has the value true if all elements of *mask* are true.
2. If *dim* is present or *mask* has rank two or greater, the result is an array of rank *n*-1 and of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ 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

```
integer, dimension (2,3) :: a, b
logical, dimension (2) :: c
logical, dimension (3) :: d
logical :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                ! represents |1 3 5|
                             |2 4 6|
b = reshape((/1,2,3,5,6,4/), (/2,3/))
                ! represents |1 3 6|
                             |2 5 4|
e = all(a==b)   ! e is assigned the value false
d = all(a==b, 1)! d is assigned the value true,false,
                ! false
c = all(a==b, 2)! c is assigned the value false,false
```

ALLOCATABLE Statement

Description

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

Syntax

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

Where:

array-name is the name of an array.

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

Remarks

The *array-name* must not be a dummy argument or a function result.

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

Example

```
integer :: a, b, c(:,:,:) ! rank of c is specified
dimension b(:,:)          ! rank of b is specified
allocatable a(:), b, c    ! rank of a is specified,
                          ! a,b, and c are allocatable
allocate (a(2), b(3,-1:1), c(10,10,10))
                          ! shapes specified,
                          ! space allocated
   ...
deallocate (a,b,c)        ! space deallocated
```

ALLOCATE Statement

Description

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.

Syntax

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

Where:

allocation-list is a comma-separated list of pointers or allocatable arrays and, for each allocatable array, 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

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 executable program is terminated.

For an allocatable array:

1. Subsequent redefinition of *lower-bound* or *upper-bound* does not affect the array bounds.
2. If *lower-bound* is omitted, the default value is one.
3. If *upper-bound* is less than *lower-bound*, the extent of that dimension is zero and the array has zero size.
4. The allocatable array can be of type CHARACTER with zero length.
5. Allocating a currently allocated allocatable array causes an error condition in the ALLOCATE statement.
6. The ALLOCATED intrinsic function can be used to determine whether an allocatable array is currently allocated.

For a pointer:

1. 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.
2. The ASSOCIATED intrinsic function can be used to determine whether a pointer is currently associated with a target.
3. A function whose result is a pointer must cause the pointer to be associated or dissociated.

Example

```
logical :: l,m
integer, pointer :: i
integer, allocatable, dimension (:,:) :: j
l = associated (i)  ! l is assigned the value false
m = allocated (j)   ! m is assigned the value false
allocate (j(4,-2:3))! shape of J defined,
                    ! space allocated
allocate (i)        ! i points to unnamed target
l = associated (i)  ! l is assigned the value true
m = allocated (j)   ! m is assigned the value true
   ...
deallocate (i,j)    ! space deallocated
```

ALLOCATED Function

Description

Indicate whether an allocatable array has been allocated.

Syntax

ALLOCATED (*array*)

Arguments

array must be an allocatable array.

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. The result is undefined if the allocation status of *array* is undefined.

Example

```
integer, allocatable :: i(:,:)
allocate (i(2,3))
l = allocated (i) ! l is assigned the value true
```

ANINT Function

Description

REAL representation of the nearest whole number.

Syntax

ANINT (*a*, *kind*)

Required Arguments

a must be of type REAL.

Optional Arguments

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

Result

The result is of type REAL. If *kind* is present, the kind is that specified by *kind*; otherwise, it is the kind of *a*. 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

```
x = anint (7.73)  ! x is assigned the value 8.0
```

ANY Function

Description:

Determine whether any values are true in a mask along a given dimension.

Syntax

ANY (*mask*, *dim*)

Required Arguments

mask must be of type LOGICAL. It must not be scalar.

Optional Arguments

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

Result

The result is of type LOGICAL with the same kind as mask. Its value and rank are computed as follows:

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

Example

```
integer, dimension (2,3) :: a, b
logical, dimension (2) :: c
logical, dimension (3) :: d
logical :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                !  represents  |1 3 5|
                               |2 4 6|
b = reshape((/1,2,3,5,6,4/), (/2,3/))
                !  represents  |1 3 6|
                               |2 5 4|
e = any(a==b)   ! e is assigned the value true
d = any(a==b, 1)! d is assigned the value true, true,
                ! false
c = any(a==b, 2)! c is assigned the value true, true
```

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 next 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.

Example

```
if (b) 10,20,30 ! goto 10 if b<0
                ! goto 20 if b=0
                ! goto 30 if b>0
```

ASIN Function

Description
Arcsine.

Syntax
ASIN (*x*)

Arguments
x must be of type REAL and must be in the range $-1 \le x \le 1$.

Result
The result has the same kind as *x*. Its value is a REAL representation of the arcsine of *x*, expressed in radians.

Example
```
r = asin(.5)  ! r is assigned the value 0.523599
```

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.

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

ASSIGN Statement (obsolescent)

Description

Assigns a statement 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.

Example

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

Assignment Statement

Description

Assigns the value of the expression on the right side of the equal sign to the variable on the left side of the equal sign.

Syntax

variable = *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; a CHARACTER variable can only be assigned a CHARACTER with the same kind; a LOGICAL variable can only be assigned a LOGICAL; and a derived type variable can only be assigned the same derived type.

Evaluation of *expression* takes place before the assignment. 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*. Precision can be lost.

If *expression* is array-valued, 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

```
real :: a=1.5, b(10)
integer :: i=2, j(10)
character (len = 5) :: string5 = "abcde"
character (len = 7) :: string7 = "cdefghi"
type person
     integer :: age
     character (len = 25) :: name
end type person
type (person) :: person1, person2
i = a             ! i is assigned int(a)
i = j             ! error
j = i             ! each element in j assigned
                  ! the value 2
j = b             ! each element in j assigned
                  ! corresponding value in b
                  ! converted to integer
string5 = string7 ! string5 is assigned "cdefg"
string7 = string5 ! string7 is assigned "abcde  "
person1 % age = 5
person1 % name = "john"
person2 = person1 ! each component of person2 is
                  ! assigned the value of the
                  ! corresponding component
                  ! of person1
```

ASSOCIATED Function

Description

Indicate whether a pointer is associated with a target.

Syntax

ASSOCIATED (*pointer*, *target*)

Required Arguments

pointer must be a pointer whose pointer association status is not undefined.

Optional Arguments

target must be a pointer or target. If it is a pointer, its pointer association status must not be undefined.

Result

The result is of type default LOGICAL. If *target* is absent, the result is true if *pointer* is currently associated with a target and false if it is not. If *target* is present and is a target, the result is true if *pointer* is currently associated with *target* and false if it is not. If *target* is present and is a pointer, the result is true if both *pointer* and *target* are currently associated with the same target and false if they are not.

Example

```
real, pointer :: a, b, e
real, target :: c, f
logical :: l
a => c
b => c
e => f
l = associated (a)    ! l is assigned the value true
l = associated (a, c) ! l is assigned the value true
l = associated (a, b) ! l is assigned the value true
l = associated (a, f) ! l is assigned the value false
l = associated (a, e) ! l is assigned the value false
```

ATAN Function

Description

Arctangent.

Syntax

ATAN (*x*)

Arguments

x must be of type REAL.

Result

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

Example

```
a = atan(.5)  ! a is assigned the value 0.463648
```

ATAN2 Function

Description

Arctangent of *y*/*x* (principal value of the argument of the complex number (*x*,*y*)).

Syntax

ATAN2 (*y*, *x*)

Arguments

y must be of type REAL.

x must be of the same kind as *y*. If *y* has the value zero, *x* must not have the value zero.

Result

The result is of the same kind as *x*. Its value is a REAL representation, expressed in radians, of the argument of the complex number (*x*,*y*).

Example

```
x = atan2 (1, 1) ! x is assigned the value 0.785398
```

BACKSPACE Statement

Description

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

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.

Example

```
backspace 10  ! file connected to unit 10 backspaced
backspace (10, err = 100)
              ! file connected to unit 10 backspaced
              ! on error goto label 100
```

BIT_SIZE Function

Description

Size, in bits, of a data object of type INTEGER.

Syntax

BIT_SIZE (*i*)

Arguments

i must be of type INTEGER.

Result

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

Example

```
integer :: i, m
integer, dimension (2) :: j, n
m = bit_size (i) ! m is assigned the value 32
n = bit_size (j) ! n is assigned the value [32 32]
```

BLOCK DATA Statement

Description

The BLOCK DATA statement begins a block data program unit.

Syntax

BLOCK DATA *[block-data-name]*

Where:

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

Example

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

BREAK Subroutine

Description

Handle break interrupts during execution of the program.

Syntax

BREAK (*lvar*)

Optional Arguments

lvar must be of type LOGICAL. It must have the SAVE attribute or be in a common block.

Remarks

If *lvar* is absent, the program will terminate after a `<Ctrl-Break>` or `<Ctrl-C>` is typed at the keyboard. All file buffers will be flushed, and the program will terminate with an error status. This is the system default action.

If *lvar* is present, the program will not terminate after a `<Ctrl-Break>` or `<Ctrl-C>`, but *lvar* will be assigned the value true. If a break is received during console input/output, some data may be lost and an error may result. The error may be trapped using the ERR= or IOSTAT= specifier in the input/output statement.

To ignore break interrupts in the program use the NBREAK subroutine (see *"NBREAK Subroutine"* beginning on page 177).

Example

```
call break ()      ! break interrupt terminates program
call break (lvar) ! break interrupt assigns true to lvar
```

BTEST Function

Description

Test a bit of an INTEGER data object.

Syntax

BTEST (*i*, *pos*)

Arguments

i must be of type INTEGER.

pos must be 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.

Result

The result is of type default LOGICAL. It has the value true if bit *pos* has the value 1 and false if bit *pos* has the value zero.

Example

```
l = btest (1, 0)   ! l is assigned the value true
l = btest (4, 1)   ! l is assigned the value false
l = btest (32, 5)  ! l is assigned the value true
```

CALL Statement

Description

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

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 be a pointer and the types, type parameters, and ranks must agree. 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.

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 are not permitted 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

```
...
call alpha (x, y)
...
subroutine alpha (a, b)
  impicit none
  real, intent(in) :: a
  real, intent(out) :: b
  ...
end subroutine alpha
```

CARG Function

Description

Pass *item* to a procedure as a C data type by value. CARG can only be used as an actual argument.

Syntax

CARG (*item*)

Arguments

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

Result

The result is the value of *item*. Its C data type is as follows:

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	double
REAL	8	double
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;};
COMPLEX	8	must not be passed by value; if passed by reference (without CARG) it is a pointer to a structure of the form: struct dp_complex { double real_part; double imaginary_part;};
LOGICAL	1	unsigned char
LOGICAL	4	must not be passed by value or by reference
CHARACTER	1	char *

Example

```
i = my_c_function(carg(a)) ! a is passed by value
```

CASE Construct

Description

The CASE construct is used to select between blocks of executable code based on the value of an expression.

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 (see SELECT CASE). 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. 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.*

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.

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

The ranges of *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 be identified by a *construct-name*. If a CASE statement is identified by a *construct-name*, the corresponding SELECT CASE statement must specify the same *construct-name*.

Example

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

CASE Statement

Description

Execution of a SELECT CASE statement causes the case expression to be evaluated (see SELECT CASE). 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. 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*.

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.

Syntax

CASE (*case-selector [*, *case-selector] ...*) *[construct-name]*

or

CASE DEFAULT *[construct-name]*

Where:

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.

construct-name is an optional name assigned to the construct.

Remarks

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

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

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

If a CASE statement is identified by a *construct-name*, the corresponding SELECT CASE statement must specify the same *construct-name*.

Example

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

CEILING Function

Description

Smallest INTEGER greater than or equal to a number.

Syntax

CEILING (*a*, *kind*)

Required Arguments

a must be of type REAL.

Optional Arguments

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

Result

The result is an INTEGER whose value is the smallest integer greater than or equal to *a*. If *kind* is present, the kind is that specified by *kind*. If *kind* is absent, the kind is that of the default REAL type.

Example

```
i = ceiling (-4.7)  ! i is assigned the value -4
i = ceiling (4.7)   ! i is assigned the value 5
```

CHAR Function

Description

Given character in the collating sequence of a given character set.

Syntax

CHAR (*i*, *kind*)

Required Arguments

i must be of type INTEGER. It 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 must be a scalar INTEGER expression that can be evaluated at compile time.

Result

The result is a CHARACTER of length one corresponding to the *i*th character of the given character set. If *kind* is present, the kind is that specified by *kind*. If *kind* is absent, the kind is that of the default CHARACTER type.

Example

```
c = char(65)  ! char is assigned the value 'A'
              ! with ASCII the default character type
```

CHARACTER Statement

Description

The CHARACTER statement declares entities of type CHARACTER.

Syntax

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

Where:

char-selector is *length-selector*
or (LEN = *type-param*, KIND = *kind-param*)
or (*type-param*, KIND = *kind-param*)
or (KIND = *kind-param*, LEN = *type-param*,)

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

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

type-param is *specification-expr*
or *

specification-expr is a scalar INTEGER expression that can be evaluated on entry to the program unit.

kind-param 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.

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 a data object 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*.

The same attribute must not appear more than once in a CHARACTER statement.

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.

The ALLOCATABLE attribute can be used only when declaring an array that is not a dummy argument or a function result.

An array declared with a POINTER or an ALLOCATABLE attribute must be specified with a deferred shape.

An array-spec for a function-name that does not have the POINTER attribute must be specified with an explicit shape.

An array-spec for a function-name that does have the POINTER attribute must be specified with a deferred shape.

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

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

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

The INTENT and OPTIONAL attributes can be specified only for dummy arguments.

An entity must 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 attribute if it has the INTRINSIC attribute.

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

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

An entity must not be given explicitly any 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 (len=2) :: x,y,z     ! x,y,z of length 2
character(len = *) :: d        ! length of dummy d
                               ! determined when
                               ! procedure invoked
```

CLOSE Statement

Description

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

Syntax

CLOSE (*close-spec-list*)

Where:

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

close-spec is *[* UNIT = *] external-file-unit*
or IOSTAT = *iostat*
or ERR = *label*
or STATUS = *status*

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

iostat is a scalar default INTEGER variable. If present, it is assigned the number of the error message generated at runtime if an error occurs in executing the CLOSE statement and the program is not terminated; if no error occurs it is assigned the value zero.

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

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

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

STATUS = 'KEEP' must not be specified for a file whose status prior to execution of a CLOSE statement 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 will not exist after execution of the CLOSE statement.

Example

```
close (8, status = 'keep')  ! unit 8 closed and kept
close (err = 200, unit = 9) ! unit 9 closed; if error
                            ! occurs, branch to label
                            ! 200
```

CMPLX Function

Description

Convert to type COMPLEX.

Syntax

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

Required Arguments

x must be of type REAL, INTEGER, or COMPLEX.

Optional Arguments

y must be of type REAL or INTEGER. If *x* is of type COMPLEX, *y* must not be present.

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

Result

The result is of type COMPLEX. If *kind* is present the result is of kind *kind*; otherwise, it is of default kind. The value of the result is the complex number whose real part has the value of *x*, if *x* is an INTEGER or a REAL; whose real part has the value of the real part of *x*, if *x* is of type COMPLEX; and whose imaginary part has the value of *y*, if present, and zero otherwise.

Example

```
y = cmplx (3.2, 4.7)  ! y is assigned (3.2, 4.7)
z = cmplx (3.2)       ! z is assigned (3.2, 0.0)
```

COMMON Statement

Description

The COMMON statement provides a global data facility. It specifies blocks of physical storage, called common blocks, that can be accessed by any scoping unit in an executable program.

Syntax

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

Where:

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

common-object-list is a comma-separated list of data objects that are declared to be 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 named *common-name*. If *common-name* is omitted, all data objects in the first *common-object-list* are specified to be in blank common.

For each common block, a storage sequence is formed of storage sequences of all data objects in the common block, in the order they appear in *common-object-list*s in the scoping unit. If any storage sequence is associated by equivalence association with the storage sequence of the common block, the sequence can be extended only by adding storage units beyond the last storage unit.

Within an executable program, the storage sequences of all common blocks with the same name (or all blank commons) have the same first storage unit. This results in the association of objects in different scoping units.

A blank common has the same properties as a named common, except:

1. Execution of a RETURN or END statement can 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 a named common can be initially defined in a DATA or type declaration statement 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 scoping unit. In such case, the *common-object-list* is treated as a continuation of the *common-object-list* for that common block.

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

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.

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

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

A pointer must only become associated with pointers of the same type, kind, length, and rank.

Default-type, non-pointer data objects must only become associated with default-type, non-pointer data objects.

Non-default-type, non-pointer intrinsic data objects must only become associated with non-default-type, non-pointer intrinsic data objects.

Default CHARACTER data objects must not become associated with default REAL, DOUBLE PRECISION, INTEGER, COMPLEX, DOUBLE COMPLEX, or LOGICAL data objects.

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.

Example

```
common /first/ a,b,c       ! a, b, and c are in named
                           ! common first
common d,e,f, /second/, g  ! d, e, and f are in blank
                           ! common, g is in named
                           ! common second
common /first/ h           ! h is also in first
```

COMPLEX Statement

Description

The COMPLEX statement declares entities of type COMPLEX.

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.

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.

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 a data object being declared.

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

Remarks

The same attribute must not appear more than once in a COMPLEX statement.

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

= *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.

= *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.

The ALLOCATABLE attribute can be used only when declaring an array that is not a dummy argument or a function result.

An array declared with a POINTER or an ALLOCATABLE attribute must be specified with a deferred shape.

An *array-spec* for a *function-name* that does not have the POINTER attribute must be specified with an explicit shape.

An *array-spec* for a *function-name* that does have the POINTER attribute must be specified with a deferred shape.

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

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

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

The INTENT and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* must 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 attribute if it has the INTRINSIC attribute.

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

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

An *entity* must not be given explicitly any attribute more than once in a scoping unit.

Example

```
complex :: a, b, c    ! a, b, and c are of type complex
complex, dimension (2, 4) :: d
                      ! d is a 2 by 4 array of complex
complex :: e = (2.0, 3.14159)
                      ! complex e is initialized
```

Computed GOTO Statement

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.

Example

```
      goto (10,20,30) i
10    a = a+1  ! if i=1 control transfers here
20    a = a+1  ! if i=2 control transfers here
30    a = a+1  ! if i=3 control transfers here
```

CONJG Function

Description

Conjugate of a complex number.

Syntax

CONJG (*z*)

Arguments

z must be of type COMPLEX.

Result

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

Example

```
x = conjg (2.1, -3.2)  ! x is assigned
                       ! the value (2.1, 3.2)
```

CONTAINS Statement

Description

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

Syntax

CONTAINS

Remarks

The CONTAINS statement is not executable.

Internal procedures cannot contain other internal procedures.

Example

```
subroutine outside (a)
  implicit none
  real, intent(in) :: a
  integer :: i, j
  real :: x
  ...
  call inside (i)
  x = sin (3.89)        ! not the intrinsic sin()
  ...
  contains
  subroutine inside (k) ! not available outside outside()
    implicit none
    integer, intent(in) :: k
    ...
  end subroutine inside
```

```
  function sin (m) ! not available outside outside()
    implicit none
    real :: sin
    real, intent(in) :: m
    ...
  end function sin
end subroutine outside
```

CONTINUE Statement

Description

Execution of a CONTINUE statement has no effect.

Syntax

CONTINUE

Example

```
      do 10 i=1,100
      ...
10    continue
```

COS Function

Description

Cosine.

Syntax

COS (*x*)

Arguments

x must be of type REAL or COMPLEX.

Result

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

Example

```
r = cos(.5)  ! r is assigned the value 0.877583
```

COSH Function

Description

Hyperbolic cosine.

Syntax

COSH (*x*)

Arguments

x must be 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

```
r = cosh(.5)  ! r is assigned the value 1.12763
```

COUNT Function

Description

Count the number of true elements in a mask along a given dimension.

Syntax

COUNT (*mask*, *dim*)

Required Arguments

mask must be of type LOGICAL. It must not be scalar.

Optional Arguments

dim must be a 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. If *dim* is absent or *mask* has rank one, the result is scalar. The result is the number of elements for which *mask* is true.

2. If *dim* is present or *mask* has rank two or greater, the result is an array of rank *n*-1 and of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$ 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
integer, dimension (2) :: c
integer, dimension (3) :: d
integer :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                  !  represents  |1 3 5|
                                 |2 4 6|
b = reshape((/1,2,3,5,6,4/), (/2,3/))
                  !  represents  |1 3 6|
                                 |2 5 4|
e = count(a==b)   ! e is assigned the value 3
d = count(a==b, 1)! d is assigned the value 2,1,0
c = count(a==b, 2)! c is assigned the value 2,1
```

CPU_TIME Subroutine

Description

Processor Time.

Syntax

CPU_TIME (*time*)

Required Arguments

time must be a scalar REAL. It is an INTENT (OUT) argument that is assigned the processor time in seconds. Note that CPU_TIME only reflects the actual CPU time when the application is compiled for Windows and run on NT or when the application is compiled for extended DOS and run from DOS (not from a DOS box of Windows). Otherwise, CPU_TIME behaves like SYSTEM_CLOCK.

Example

```
call cpu_time(start_time)
x = cos(2.0)
call cpu_time(end_time)
cos_time = end_time - start_time
! time to calculate and store the cosine of 2.0
```

CSHIFT Function

Description

Circular shift of all rank one sections in an array. Elements shifted out at one end are shifted in at the other. Different sections can be shifted by different amounts and in different directions by using an array-valued shift.

Syntax

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

Required Arguments

array can be of any type. It must not be scalar.

shift must be of type INTEGER and must be scalar if *array* is of rank one; otherwise it must be scalar or 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*.

Optional Arguments

dim must be a 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 array-valued.

Example

```
integer, dimension (2,3) :: a, b
integer, dimension (3) :: c, d
integer :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                  !  represents  |1 3 5|
                                 |2 4 6|
c = (/1,2,3/)
b = cshift(a,1)   ! b is assigned the value |2 4 6|
                  !                         |1 3 5|
b = cshift(a,-1,2)! b is assigned the value |3 5 1|
                  !                         |4 6 2|
b = cshift(a,c,1) ! b is assigned the value |2 3 5|
                  !                         |1 4 6|
d = cshift(c,2)   ! c is assigned the value |3 1 2|
```

CYCLE Statement

Description

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

Syntax

CYCLE *[do-construct-name]*

Where:

do-construct-name is the name of a DO construct that contains the CYCLE statement. 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.

Example

```
outer: do i=1, 10
inner:   do j=1, 10
           if (i>a) cycle outer
           if (j>b) cycle ! cycles to inner
           ...
         enddo inner
       enddo outer
```

DATA Statement

Description

The DATA statement provides initial values for variables.

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

object-list is expanded to form a sequence of scalar variables. An array whose unqualified name 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.

value-list is expanded to form a sequence of scalar constant values. Each such value must be a constant that is either previously defined or made accessible by a use association or host association. *repeat* indicates the number of times the following constant is to be included in the sequence; omission of *repeat* has the effect of a repeat factor of 1.

The expanded sequences of scalar variables and constant values are in one-to-one correspondence. Each constant specifies the initial value for the corresponding variable. The lengths of the two expanded sequences must be the same.

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

A variable whose name is included in an *object-list* must not be: a dummy argument made accessible by use association or host association; in a named common block unless the DATA statement is in a block data program unit; in a blank common block; a function name; a function result name; an automatic object; a pointer; or an allocatable array.

In an array element or a scalar structure component that is in an *implied-do-object-list*, any subscript must be an expression whose primaries are either constants or *implied-do-var*s of the containing *implied-do*s, and each operation must be intrinsic.

expr must involve as primaries only constants or *implied-do-var*s of the containing *implied-do*s, and each operation must be intrinsic.

The value of the constant must be compatible with its corresponding variable according to the rules of intrinsic assignment, and the variable becomes initially defined with the value of the constant in accordance with the rules of intrinsic assignment.

Example

```
real :: a
integer, dimension (-3:3) :: smallarray
integer, dimension (10000) :: bigarray
data a /3.78/, smallarray /7 * 1/
                    ! assigns 3.78 to a and 1 to each
                    ! element of smallarray
data (bigarray(i), i=1,10000,2) /5000*6/
                    ! assigns 6 to each element that
                    ! has an odd subscript value
```

DATE_AND_TIME Subroutine

Description

Date and real-time clock data.

Syntax

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

Optional Arguments

date must be scalar and of type default CHARACTER, and must be of length at least eight in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost eight characters are set to a value of the form *ccyymmdd*, where *cc* is the century, *yy* the year within the century, *mm* the month within the year, and *dd* the day within the month. If there is no date available, they are set to blank.

time must be scalar and of type default CHARACTER, and must be of length at least ten in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost ten characters are set to a value of the form *hhmmss.sss*, where *hh* is the hour of the day, *mm* is the minutes of the hour, and *ss.sss* is the seconds and milliseconds of the minute. If there is no clock available, they are set to blank.

zone must be scalar and of type default CHARACTER, and must be of length at least five in order to contain the complete value. It is an INTENT (OUT) argument. Its leftmost five characters are set to a value of the form +-*hhmm*, where *hh* and *mm* are the time difference with respect to Coordinated Universal Time (UTC, also known as Greenwich Mean Time) in hours and parts of an hour expressed in minutes, respectively. If there is no clock available, they are set to blank. To use the zone argument, you must first set the environment variable TZ as follows:

`set TZ=`*ZZZ[+/-]d[d][LLL]*

where *ZZZ* is a three-character string representing the name of the current time zone; *[+/-]d[d]* is a required field containing an optionally signed number with one or two digits representing the local time zone's difference from UTC in hours (negative numbers adjust eastward from UTC); and *[LLL]* is an optional three-character field that represents the local time zone's daylight savings time. If *[LLL]* is present then 1 is added to *[+/-]d[d]*. *ZZZ* and *LLL* (if present) must be uppercase. For example, "TZ=PST-8PDT" would be used on the west coast of the United States during the portion of the year when daylight savings is in effect, and "TZ=PST-8" during the rest of the year. If the TZ environment variable is not set or is set using an invalid format then zone will be set to blanks.

values must be of type default INTEGER and of rank one. It is an INTENT (OUT) argument. Its size must be at least eight. The values returned in VALUES are as follows:

values (1) the year (for example, 1990), or `-huge(0)` if there is no date available.

values (2) the month of the year, or `-huge(0)` if there is no date available.

values (3) the day of the month, or `-huge(0)` if there is no date available.

values (4) the time difference with respect to Coordinated Universal Time (UTC) in minutes, or `-huge(0)` if this information is not available.

values (5) the hour of the day, in the range of 0 to 23, or `-huge(0)` if there is no clock.

values (6) the minutes of the hour, in the range of 0 to 59, or `-huge(0)` if there is no clock.

values (7) the seconds of the minute, in the range 0 to 60, or `-huge(0)` if there is no clock.

values (8) the milliseconds of the second, in the range 0 to 999, or `-huge(0)` if there is no clock.

Example

```
! called in Incline Village, NV on February 3, 1993
! at 10:41:04.1
integer :: dt(8)
character (len=10) :: time, date, zone
call date_and_time (date, time, zone,  dt)
! date is assigned the value "19930203"
! time is assigned the value "104104.100"
! zone is assigned the value "-800"
! dt is assigned the  value: 1993,2,3,
!                            -480,10,41,4,100.
```

DBLE Function

Description

Convert to double-precision REAL type.

Syntax

DBLE (*a*)

Arguments

a must be 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

```
double precision d
d = dble (1)  ! d is assigned the value 1.00000000000000
```

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.

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 with the TARGET attribute causes the pointer association status of any pointer associated with it to become undefined.

Example

```
deallocate (a, b, stat=s) ! causes a and b to be
                          ! deallocated.  If success-
                          ! ful, s is assigned 0
```

Derived-Type Definition Statement

Description

The derived type definition statement begins a derived type definition.

Syntax

TYPE *[[, access-spec]* :: *] type-name*

Where:

access-spec is PUBLIC
or PRIVATE

type-name is the name of the derived type being defined.

Remarks

access-spec is permitted only if the derived type definition is within the specification part of a module.

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.

Example

```
type coordinate
  real :: x,y
end type
```

DIGITS Function

Description

Number of significant binary digits.

Syntax

DIGITS (*x*)

Arguments

x must be of type INTEGER or REAL. It can be scalar or array-valued.

Result

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

Example

```
real :: r
integer :: i
i = digits (r)  ! i is assigned the value 24
```

DIM Function

Description

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

Syntax

DIM (*x*, *y*)

Arguments

x must be of type INTEGER or REAL.

y must be of the same type and kind as *x*.

Result

The result is of the same type as *x*. Its value is *x* - *y* if *x* is greater than *y* and zero otherwise.

Example

```
z = dim(1.1, 0.8) ! z is assigned the value 0.3
z = dim(0.8, 1.1) ! z is assigned the value 0.0
```

DIMENSION Statement

Description

The DIMENSION statement specifies the shape of an array.

Syntax

DIMENSION *[::] array-name* (*array-spec*) *[, array-name* (*array-spec*) *] ...*

Where:

array-name is the name of an array.

array-spec is *explicit-shape-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.

Example

```
dimension a(3,2,1) ! a is a 3x2x1 array
dimension b(-3:3)  ! b is a 7-element vector with a
                   ! lower bound of -3
dimension c(:,:,:) ! c is an assumed-shape or
                   ! deferred-shape array of
                   ! rank 3
dimension d(*)     ! d is an assumed-size array
```

DLL_EXPORT Statement

Description

The DLL_EXPORT statement specifies which procedures should be available in a dynamic-link library.

Syntax

DLLEXPORT *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.

Example

```
function half(x)
  implicit none
  integer :: half
  dll_export half
  half = x/2
  return
end function half
```

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.

Example

```
program main
  implicit none
  integer :: foo, i
  dll_import foo
  i = half(i)
  stop
end program main
```

DO Construct

Description

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

Syntax

do-statement
block
do-termination

Where:

do-statement is a DO statement

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 statement is an action statement other than a GOTO, RETURN, STOP, EXIT, CYCLE, assigned GOTO, arithmetic IF, or END statement.

Remarks

If a construct name is specified in the DO statement, the same construct name must be specified in a corresponding END DO statement.

Ending a DO construct with a labeled action statement is obsolescent.

Example

```
do i=1,100          ! iterates 100 times
  do while (a>b)    ! iterates while a>b
    do 10 j=1,100,3 ! iterates 33 times
      ...
10  continue
  end do
end do
```

The CYCLE statement can be used to curtail execution of the current iteration of a DO loop. The EXIT statement can be used to exit a DO loop altogether.

DO Statement

Description

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

Syntax

[construct-name : *]* DO *[label] [loop-control]*

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, default REAL, or default double-precision REAL.

expr is a scalar expression of type INTEGER, default REAL, or default double-precision REAL. 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.

Remarks

When a DO statement is executed, a DO construct becomes active. The expressions in *loop-control* are evaluated, and, if *do-variable* is present, it is assigned an initial value and an iteration count is established for the construct based on the expressions. 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 and the DO construct becomes inactive. 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* is obsolescent.

Example

```
do i=1,100          ! iterates 100 times
  do while (a>b)    ! iterates while a>b
    do 10 j=1,100,3 ! iterates 33 times each time
                    ! this do construct is entered
      ...
10  continue
  end do
end do
```

DOT_PRODUCT Function

Description

Dot-product multiplication of vectors.

Syntax

DOT-PRODUCT (*vector_a*, *vector_b*)

Arguments

vector_a must be of type INTEGER, REAL, COMPLEX, or LOGICAL. It must be array-valued and of rank one.

vector_b must be of numeric type if *vector_a* is of numeric type and of type LOGICAL if *vector_a* is of type LOGICAL. It must be array-valued, of rank one, and of the same size as *vector_a*.

Result

If the arguments are of type LOGICAL, then the result is scalar and of type default LOGICAL. Its value is ANY (*vector_a* .AND. *vector_b*). If the vectors have size zero, the result has the value false.

If the arguments are of different numeric type, then the result type is that of the argument with the higher type, where COMPLEX is higher than REAL, and REAL is higher than INTEGER. If both arguments are of the same type, the result kind is the kind of the argument that offers the greater range. The result value is SUM (*vector_a* * *vector_b*) if *vector_a* is of type REAL or INTEGER. The result value is SUM (CONJG (*vector_a*) * *vector_b*) if *vector_a* is of type COMPLEX.

Example

```
i = dot_product((/3,4,5/),(/6,7,8/))
        ! i is assigned the value 86
```

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.

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 a data object being declared.

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

Remarks

The same attribute must not appear more than once in a DOUBLE PRECISION statement.

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.

The ALLOCATABLE attribute can be used only when declaring an array that is not a dummy argument or a function result.

An array declared with a POINTER or an ALLOCATABLE attribute must be specified with a deferred shape.

An *array-spec* for a *function-name* that does not have the POINTER attribute must be specified with an explicit shape.

An *array-spec* for a *function-name* that does have the POINTER attribute must be specified with a deferred shape.

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

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

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

The INTENT and OPTIONAL attributes can be specified only for dummy arguments.

An entity must 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 attribute if it has the INTRINSIC attribute.

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

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

An *entity* must not be given explicitly any attribute more than once in a scoping unit.

Example

```
double precision a, b, c ! a, b, and c are of type
                         ! double precision
double precision, dimension (2, 4) :: d
                         ! d is a 2 by 4 array
                         ! of double precision
double precision :: e = 2.0d0
                         ! e is initialized
```

DPROD Function

Description

Double-precision REAL product.

Syntax

DPROD (*x*, *y*)

Arguments

x must be of type default REAL.

y must be of type default REAL.

Result

The result is of type double-precision REAL. Its value is a double-precision representation of the product of *x* and *y*.

Example

```
dub = dprod (3.e2, 4.4e4) ! dub is assigned 13.2d6
```

DVCHK Subroutine

Description

The initial invocation of the DVCHK subroutine masks the divide-by-zero interrupt on the floating-point unit. *lflag* must be set to true on the first invocation. Subsequent envocations 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.

Syntax

DVCHK (*lflag*)

Arguments

lflag must be of type LOGICAL. It is assigned the value true if a divide-by-zero exception has occurred, and false otherwise.

Example

```
call dvchk (lflag) ! mask the divide-by-zero interrupt
```

ELSE IF Statement

Description

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

Syntax

ELSE IF (*expr*) THEN *[construct-name]*

Where:

expr is a scalar LOGICAL expression.

construct-name is the optional name given to the IF construct.

Example

```
if (i>-1) then
  print*, b
else if (i<j) then ! executed only if true and previous
                      ! if expression was false
  print*, c
end if
```

ELSE Statement

Description

The ELSE statement controls precedes a block of code to be executed in an IF construct where all previous IF expressions are false.

Syntax

ELSE *[construct-name]*

Where:

construct-name is the optional name given to the IF construct.

Example

```
if (i>j) then
  print*, a
else if (i<j) then
  print*, b
else   ! executed if previous if expressions were false
  print*, c
end if
```

ELSEWHERE Statement

Description

The ELSEWHERE statement controls conditional execution of a block of assignment statements for elements of an array for which the WHERE construct's mask expression is false.

Syntax

ELSEWHERE

Remarks

In each assignment statement the mask expression and the variable on the left side of the assignment statement must be of the same shape.

The assignment statement must not be a defined assignment

Example

```
where (b>c)         ! begin where construct
  b = -1
elsewhere
  b = 1
end where
```

END Statement

Description

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

Syntax

END *[class [name]]*

Where:

class is either PROGRAM, FUNCTION, SUBROUTINE, MODULE, INTERFACE or BLOCK DATA.

name is the name of the program unit, module subprogram, or internal subprogram.

Remarks

Each program unit, module subprogram, or internal subprogram must have exactly one END statement.

The END PROGRAM, END FUNCTION, and END SUBROUTINE statements are executable and can be branch target statements. The END MODULE, END INTERFACE, and END BLOCK DATA statements are non-executable.

Executing an END FUNCTION or END SUBROUTINE statement is equivalent to executing a return statement in a subprogram.

Executing an END PROGRAM statement terminates the executing program.

name can be used only if a name was given to the program unit, module subprogram, or internal subprogram in a PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement. *name* cannot be used with an END INTERFACE statement.

If *name* is present, it must be identical to the *name* specified in the PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement.

Example

```
program names
  call joe
  call bill
  call fred
end program names   ! program and names are optional

subroutine joe
end subroutine joe  ! ok end statement

subroutine bill
end subroutine      ! also ok end statement

subroutine fred
end                 ! also ok end statement
```

END DO Statement

Description

The END DO statement ends a DO construct.

Syntax

END DO *[construct-name]*

Where:

construct-name is the name of the DO construct.

Remarks

If the DO statement of the DO construct is identified by a *construct-name*, the corresponding END DO statement must specify the same *construct-name*. If the DO statement is not identified by a *construct-name*, the END DO statement must not specify a *construct-name*.

If the DO statement specifies a label, the corresponding END DO statement must be identified with the same label.

Example

```
named: do i=1,10
labeled:  do 10 j=1,10
             do k=1,10
               ...
             end do
10        end do labeled
       end do named
```

ENDFILE Statement

Description

The ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record, which becomes the last record of the file.

Syntax

ENDFILE *unit-number*

or

ENDFILE (*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. If *stat* is present and error, end-of-file, or end-of-record condition occurs, execution is not terminated.

Remarks

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be executed to reposition the file before any data transfer statement or subsequent ENDFILE statement.

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.

Example

```
endfile 8  ! writes an endfile record to the file
           ! connected to unit 8
```

END IF Statement

Description

The END IF statement ends an IF construct.

Syntax

END IF *[construct-name]*

Where:

construct-name is the name of the IF construct.

Remarks

If the IF statement of the IF construct is identified by a *construct-name*, the corresponding END IF statement must specify the same *construct-name*. If the IF statement is not identified by a *construct-name,* the END IF statement must not specify *construct-name*.

Example

```
if (a.gt.b) then
  c = 1
  d = 2
end if
```

END SELECT Statement

Description

The END SELECT statement ends a CASE construct.

Syntax

END SELECT *[construct-name]*

Where:

construct-name is the name of the CASE construct.

Remarks

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

Example

```
select case (i)
case (:-1)
  print*, "negative"
case (0)
  print*, "zero"
case (1:)
  print*, "positive"
end select
```

END WHERE Statement

Description

The END WHERE statement ends a WHERE construct.

Syntax

END WHERE

Example

```
where (c > d)  ! c and d are arrays
  c = 1
  d = 2
end where
```

ENTRY Statement

Description

The ENTRY statement permits one 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 the name of the result.

Remarks

An ENTRY statement can appear only in an external subprogram or module subprogram. An ENTRY statement must not appear within an executable construct.

ENTRY statement in a function

If the ENTRY statement is contained in a function subprogram, an additional function is defined by that subprogram. The name of the function is *entry-name* and its result variable is *result-name* or is *entry-name* if no *result-name* is provided. The characteristics of the function result are specified by specifications of the result variable.

If RESULT is specified, *entry-name* must not appear in any specification statement in the scoping unit of the function program.

RESULT can be present only if the ENTRY statement is contained in a function subprogram.

If RESULT is specified, *result-name* must not be the same as *entry-name*.

ENTRY statement in a subroutine

A dummy argument can be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

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 the subroutine are those specified on the ENTRY statement.

Example

```
program main
  i=2
  call square(i)
  j=2
  call quad(j)
  print*, i,j   ! prints 4         16
end program main
subroutine quad(k)
  k=k*k
entry square(k)
  k=k*k
  return
end subroutine quad
```

EOSHIFT Function

Description

End-off shift of all rank one sections in an array. Elements are shifted out at one end and copies of boundary values are shifted in at the other. Different sections can be shifted by different amounts and in different directions by using an array-valued shift.

Syntax

EOSHIFT (*array*, *shift*, *boundary*, *dim*)

Required Arguments

array can be of any type. It must not be scalar.

shift must be of type INTEGER and must be scalar if *array* is of rank one; otherwise it must be scalar or 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*.

Optional Arguments

boundary must be of the same type and kind as *array*. If *array* is of type CHARACTER, *boundary* must have the same length as *array*. It must be scalar if *array* is of rank one; otherwise it must be scalar or of rank *n*-1 and of shape $(d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n)$. *boundary* can be omitted, in which case the default values are zero for numeric types, blanks for CHARACTER, and false for LOGICAL.

dim must be a scalar INTEGER with a value in the range $1 \leq dim \leq n$, where *n* is the rank of *array*. If *dim* is omitted, it is as if it were present with a value of one.

Result

The result is of the same type, kind and shape as *array*.

Element $(s_1, s_2, \ldots, s_n)$ of the result has the value *array* $(s_1, s_2, \ldots, s_{dim-1}, s_{dim} + sh, s_{dim+1}, \ldots, s_n)$ where *sh* is *shift* or *shift* $(s_1, s_2, \ldots, s_{dim-1}, s_{dim+1}, \ldots, s_n)$ provided the inequality $lbound(array, dim) \leq s_{dim} + sh \leq ubound(array, dim)$ holds and is otherwise *boundary* or *boundary* $(s_1, s_2, \ldots, s_{dim-1}, s_{dim+1}, \ldots, s_n)$.

Example

```
integer, dimension (2,3) :: a, b
integer, dimension (3) :: c, d
integer :: e
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                   !  represents  |1 3 5|
                                  |2 4 6|
c = (/1,2,3/)
b = eoshift(a,1)   ! b is assigned the value |0 0 0|
                   !                         |1 3 5|
b = eoshift(a,-1,0,2) ! b assigned the value |3 5 0|
                   !                         |4 6 0|
b = eoshift(a,-c,1)! b is assigned the value |2 1 1|
                   !                         |1 1 1|
d = eoshift(c,2)   ! c is assigned the value |3 0 0|
```

EPSILON Function

Description

Positive value that is almost negligible compared to unity.

Syntax

EPSILON (*x*)

Arguments

x must be of type REAL. It can be scalar or array-valued.

Result

The result is a scalar value of the same kind as *x*. Its value is 2^{1-p}, where *p* is the number of bits in the fraction part of the physical representation of *x*.

Example

```
! reasonably safe compare of two default REALs
function equals (a, b)
  implicit none
  logical :: equals
  real, intent(in) :: a, b
  real :: eps
  eps = abs(a) * epsilon(a)  ! scale epsilon
  if (eps == 0) then
    eps = tiny (a)        ! if eps underflowed to 0
                          ! use a very small
                          ! positive value for epsilon
  end if
  if (abs(a-b) > eps) then
    equals = .false.      ! not equal if difference>eps
    return
  else
    equals = .true.       ! equal otherwise
    return
  endif
end function equals
```

EQUIVALENCE Statement

Description

The EQUIVALENCE statement is used to specify that two or more objects in a scoping unit 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

If the equivalenced objects have different types or kinds, the EQUIVALENCE statement does not cause any type conversion or imply mathematical equivalence.

If a scalar and an array-valued object are equivalenced, the scalar does not have array properties and the array does not have scalar properties.

An *equivalence-object* must not be a dummy argument, a pointer, an allocatable array, an object of a non-sequence derived type or 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 the preceding objects.

If an *equivalence-object* is of a derived type that is not a numeric sequence or CHARACTER sequence type, all of the objects in the equivalence set must be of the same type.

If an *equivalence-object* is of an intrinsic type other than default INTEGER, default REAL, double precision REAL, default COMPLEX, default LOGICAL, or default CHARACTER, all of the objects in *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 must not specify that the same storage unit is to occur more than once in a storage sequence.

Example

```
equivalence (a,b,c(2)) ! a, b, and c(2) share the
                       ! same storage
```

ERROR Subroutine

Description

Print a message to the console, then continue processing.

Syntax

ERROR (*message*)

Arguments

message must be of type CHARACTER. It is an INTENT(IN) argument that is the message to be printed. Note that to generate a subprogram traceback you must specify the -trace compiler switch.

Example

```
call error('error')        ! prints the string 'error'
                           ! followed by a subprogram
                           ! traceback
```

EXIT Statement

Description

The EXIT statement terminates a DO loop.

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, it is as if *do-construct-name* were the name of the innermost DO construct in which the EXIT statement appears.

Example

```
outer: do i=1, 10
inner:   do j=1, 10
           if (i>a) exit outer
           if (j>b) exit ! exits inner
           ...
         enddo inner
       enddo outer
```

EXIT Subroutine

Description

Terminate the program and set the DOS error level.

Syntax

EXIT (*ilevel*)

Arguments

ilevel must be of type INTEGER. It is the DOS error level set on program termination.

Example

```
call exit(3)  ! exit -- DOS error level 3
```

EXP Function

Description

Exponential.

Syntax

EXP (*x*)

Arguments

x must be of type REAL or COMPLEX.

Result

The result is of the same type as *x*. Its value is a REAL or COMPLEX representation of e^x. If *x* is of type COMPLEX, its imaginary part is treated as a value in radians.

Example

```
a = exp(2.0)  ! a is assigned the value 7.38906
```

EXPONENT Function

Description

Exponent part of the model representation of a number.

Syntax

EXPONENT (*x*)

Arguments

x must be of type REAL.

Result

The result is of type default INTEGER. Its value is the value of the exponent part of the model representation of *x*.

Example

```
i = exponent(3.8) ! i is assigned 2
i = exponent(-4.3)! i is assigned 3
```

EXTERNAL Statement

Description

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

Syntax

EXTERNAL *external-name-list*

Where:

external-name-list is a comma-separated list of external procedures, dummy procedures, or block data program units.

Remarks

If an intrinsic procedure name appears in an EXTERNAL statement, the intrinsic procedure is not available in the scoping unit and the name is that of an external procedure.

A name can appear only once in all of the EXTERNAL statements in a scoping unit.

Example

```
subroutine fred (a, b, sin)
external sin   ! sin is the name of an external
               ! procedure, not the intrinsic sin()
call bill (a, sin)
               ! sin can be passed as an actual arg
```

FLOOR Function

Description

Greatest INTEGER less than or equal to a number.

Syntax

FLOOR (*a*, *kind*)

Required Arguments

a must be of type REAL.

Optional Arguments

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

Result

The result is of type default INTEGER. Its value is equal to the greatest INTEGER less than or equal to *a*. If *kind* is present, the kind is that specified by *kind*. If *kind* is absent, the kind is that of the default REAL type.

Example

```
i = floor(-2.1)  ! i is assigned the value -3
j = floor(2.1)   ! j is assigned the value 2
```

FLUSH Subroutine

Description

Empty the buffer for an input/output unit by writing to its corresponding file. Note that this does not flush the DOS file buffer.

Syntax

FLUSH (*iunit*)

Arguments

iunit must be of type INTEGER. It is an INTENT(IN) argument that is the unit number of the file whose buffer is to be emptied.

Example

```
call flush(11)  ! empty buffer for unit 11
```

FORMAT Statement

Description

The FORMAT statement provides explicit information that directs the editing between the internal representation of data and the characters that are input or output.

Syntax

FORMAT (*[format-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 E*w.d[*E*e]*
or EN*w.d[*E*e]*
or ES*w.d[*E*e]*
or G*w.d[*E*e]*
or L*w*
or A*[w]*
or D*w.d*

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 <u>or *c*H*rep-chars*</u>

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*)

Remarks

The FORMAT statement must be labeled.

The comma between edit descriptors may be omitted in the following cases:

- between the scale factor (P) and the numberic 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.

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 edit descriptors
I (decimal INTEGER),
B (binary INTEGER),
O (octal INTEGER),
Z (hexadecimal INTEGER),
F (REAL or COMPLEX, no exponent on output),
E and D (REAL or COMPLEX, exponent on output),
EN (engineering notation),
ES (scientific notation),
G (generalized),
L (LOGICAL),
A (CHARACTER),
T (position from beginning of record),
TL (position left from current position),
TR (position right from current position),
X (position forward from current position),
S (default plus production on output),
SP (force plus production on output),
SS (suspend plus production on output),
BN (ignore non-leading blanks on input),
BZ (non-leading blanks are zeros on input),
/ (end of current record),
: (terminate format control), and
P (scale factor)
indicate the manner of data editing.

Descriptions of each edit descriptor are provided in *"Input/Output Editing"* beginning on page 24.

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 a delimiter character itself appears, either an apostrophe or quote, it must be as a consecutive pair without any blanks. Each such pair represents a single occurence of the delimiter character.

Example

```
      a = 123.45
      write (7,10) a
      write (7,20) a
10    format (e11.5)      ! 0.12345E+03
20    format (2p, e12.5)  ! 12.3450E+01
```

FRACTION Function

Description

Fraction part of the physical representation of a number.

Syntax

FRACTION (*x*)

Arguments

x must be of type REAL.

Result

The result is of the same kind as *x*. Its value is the value of the fraction part of the physical representation of *x*.

Example

```
a = fraction(3.8) ! a is assigned the value 0.95
```

FUNCTION Statement

Description

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.

Syntax

[RECURSIVE *] [type-spec]* FUNCTION *function-name* (*[dummy-arg-names]*) *[* RESULT *(result-name*)*]*

or

[type-spec] [RECURSIVE *]* 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 (*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

The keyword RECURSIVE must be present if the function directly or indirectly calls itself or a function defined by an ENTRY statement in the same subprogram. RECURSIVE must also be present if a function defined by an ENTRY statement directly or indirectly calls itself, another function defined by an ENTRY statement, or the function defined by the FUNCTION statement.

A function that calls itself directly must use the RESULT option.

If the function result is array-valued or a pointer, this must be specified in the specification of the result variable in the function body.

Example

```
integer function sum(i,j) result(k)
```

GETCL Subroutine

Description

Get command line.

Syntax

GETCL (*result*)

Arguments

result must be of type CHARACTER. It is an INTENT(OUT) argument that is assigned the characters on the DOS command line beginning with the first non-white-space character after the program name.

Example

```
call getcl(cl)  ! cl is assigned the command line
```

GETENV Function

Description

Get the specified environment variable.

Syntax

GETENV(*variable*)

Arguments

variable must be of type default CHARACTER. It is an INTENT(IN) argument which specifies the environment variable to check.

Result

The result is of type default character and is set to the value of the environment variable specified by variable. If the specified variable is not defined in the environment then GETENV will return a zero-length string.

Example

```
character (len=80) :: mypath
mypath = getenv('path')
```

GOTO Statement

Description

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

Syntax

GOTO *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
          goto 10        ! branches to 10
          b = c          ! never executed
10        c = d
```

HUGE Function

Description

Largest representable number of data type.

Syntax

HUGE (x)

Arguments

x must be 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 in the data type of *x*.

Example

```
a = huge(4.1)  ! a is assigned the value 0.340282E+39
```

IACHAR Function

Description

Position of a character in the ASCII collating sequence.

Syntax

IACHAR (*c*)

Arguments

c must be of type default CHARACTER and of length one.

Result

The result is of type default INTEGER. Its value is the position of *c* in the ASCII collating sequence and is in the range $0 \le iachar(c) \le 127$.

Example

```
i = iachar('c')  ! i is assigned the value 99
```

IAND Function

Description

Bit-wise logical AND.

Syntax

IAND (*i*, *j*)

Arguments

i must be of type INTEGER.

j must be of type INTEGER and of the same kind as *i*.

Result

The result is of type INTEGER. Its value is the value obtained by performing a bit-wise logical AND of *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

Clear one bit to zero.

Syntax

IBCLR (*i*, *pos*)

Arguments

i must be of type INTEGER.

pos must be of type INTEGER. It must be non-negative and less than the number of bits in *i*.

Result

The result is of type INTEGER and of the same kind as *i*. Its value is the value of *i* except that bit *pos* is set to zero. Note that the lowest order *pos* is zero.

Example

```
i = ibclr (37,2) ! i is assigned the value 33
```

IBITS Function

Description

Extract a sequence of bits.

Syntax

IBITS (*i*, *pos*, *len*)

Arguments

i must be of type INTEGER.

pos must be of type INTEGER. It must be non-negative and *pos*+*len* must be less than or equal to the number of bits in *i*.

len must be of type INTEGER and non-negative.

Result

The result is of type INTEGER and of the same kind as *i*. Its value is the value of the sequence of *len* bits beginning with *pos*, right adjusted with all other bits set to 0. Note that the lowest order *pos* is zero.

Example

```
i = ibits (37,2,2) ! i is assigned the value 1
```

IBSET Function

Description

Set a bit to one.

Syntax

IBSET (*i*, *pos*)

Arguments

i must be of type INTEGER.

pos must be of type INTEGER. It must be non-negative and less than the number of bits in *i*.

Result

The result is of type INTEGER and of the same kind as *i*. Its value is the value of *i* except that bit *pos* is set to one. Note that the lowest order *pos* is zero.

Example

```
i = ibset (37,1) ! i is assigned the value 39
```

ICHAR Function

Description

Position of a character in the processor collating sequence associated with the kind of the character.

Syntax

ICHAR (*c*)

Arguments

c must be of type CHARACTER and of length one.

Result

The result is of type default INTEGER. Its value is the position of *c* in the processor collating sequence associated with the kind of *c* and is in the range $0 \leq ichar(c) \leq n - 1$, where *n* is the number of characters in the collating sequence.

Example

```
i = ichar('c')  ! i is assigned the value 99 for
                ! character c in the ASCII
                ! collating sequence
```

IEOR Function

Description
Bit-wise logical exclusive OR.

Syntax
IEOR (*i*, *j*)

Arguments
i must be of type INTEGER.

j must be of type INTEGER and of the same kind as *i*.

Result
The result is of type INTEGER. Its value is the value obtained by performing a bit-wise logical exclusive OR of *i* and *j*.

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 which, if any, of one or more blocks of statements or executable constructs will be executed.

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

At most one of the blocks contained within the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks contained within the construct will be executed. The *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 or an ELSE statement 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 and there is no ELSE statement, the execution of the construct is completed without the execution of any block within the 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

```
if (a>b) then
  c = d
else if (a<b) then
  d = c
else ! a=b
  stop
end if
```

IF-THEN Statement

Description

The IF-THEN statement begins an IF construct.

Syntax

[construct-name: *]* IF (*expr*) THEN

Where:

construct-name is an optional name for the IF construct.

expr is a scalar LOGICAL expression.

Remarks

At most one of the blocks contained within the IF construct is executed. If there is an ELSE statement in the construct, exactly one of the blocks contained within the construct will be executed. The *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 or an ELSE statement 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 and there is no ELSE statement, the execution of the construct is completed without the execution of any block within the construct.

Example

```
if (a>b) then
  c = d
else
  d = c
end if
```

IF Statement

Description

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

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 *expr*. If the value of *expr* is true, *action-statement* is executed. If the value is false, *action-statement* is not executed.

Example

```
if (a >= b) a = -a
```

IMPLICIT Statement

Description

The IMPLICIT statement specifies, for a scoping unit, a type and optionally a kind or a CHARACTER length for each name beginning with a letter 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 (*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.

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

A *letter-spec* consisting of two letters separated by a minus is equivalent to writing a list containing all of the letters in alphabetical order in the alphabetic sequence from the first letter through the second letter. The 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 the null mapping for all the letters. If a mapping is not specified for a letter, the default is the mapping in the host scoping unit.

If IMPLICIT NONE is specified in a scoping unit, it must precede any PARAMETER statements that appear in the scoping unit and there must be no other IMPLICIT statements in the scoping unit.

Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic function, and is not made accessible by use association or host association is declared implicitly to be of the type (and type parameters, kind and length) mapped from the first letter of its name, provided the mapping is not null.

An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram.

Example

```
implicit character (c), integer (a-b, d-z)
                  ! specifies that all data objects
                  ! beginning with c are implicitly of
                  ! type character, and other data
                  ! objects are of type integer
```

INCLUDE Line

Description

The INCLUDE line causes text in another file to be processed as if the text therein replaced the INCLUDE line. The INCLUDE line is not a Fortran statement.

Syntax

INCLUDE *filename*

Where:

filename is a CHARACTER literal constant that corresponds to a file that contains source text to be included in place of 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.

Lahey Fortran limits the level of nesting of include files to twenty.

Example

```
include "types.for"  ! include a file named types.for
                     ! in place of this INCLUDE line
```

INDEX Function

Description

Starting position of a substring within a string.

Syntax

INDEX (*string*, *substring*, *back*)

Required Arguments

string must be of type CHARACTER.

substring must be of type CHARACTER with the same kind as *string*.

Optional Arguments

back must be of type LOGICAL.

Result

The result is of type default INTEGER. If *back* is absent or false, the result value is the position number in *string* where the first instance of *substring* begins or zero if there is no such value or if *string* is shorter than *substring*. If *substring* is of zero length, the result value is one.

If *back* is present and true, the result value is the position number in *string* where the last instance of *substring* begins. If *string* is shorter than *substring* or if *substring* is not in *string*, zero is returned. If *substring* is of zero length, LEN(*string*)+1 is returned.

Example

```
i = index('mississippi', 'si')
                 ! i is assigned the value 4
i = index('mississippi', 'si', back=.true.)
                 ! i is assigned the value 7
```

INQUIRE Statement

Description

The INQUIRE statement enables the program to make inquiries about a 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 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.

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 processor is unable to determine if sequential access is allowed for the file.

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 processor is unable to determine if direct access is allowed for the file.

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 processor is unable to determine if formatted is an allowed form for the file.

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 processor is unable to determine if unformatted is an allowed form for the file.

recl is a scalar default INTEGER variable that evaluates to the record length for a file connected for direct access, or the maximum record length for a file connected for sequential access.

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 undefined.

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.

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; and ASIS if the position is after the endfile record.

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, and READWRITE if the file is connected for input and output.

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 processor is unable to determine if READ is an allowed action on the file.

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 processor is unable to determine if WRITE is an allowed action on the file.

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 processor is unable to determine if READWRITE is an allowed action on the file.

delim is a scalar default CHARACTER variable that evaluates to APOSTROPHE if the apostrophe will be used to delimit character constants written with list-directed or namelist formatting, QUOTE if the quotation mark will be used, and NONE if neither quotation marks nor apostrophes will be used.

pad is a scalar default CHARACTER variable that evaluates to YES if the formatted input record is padded with blanks 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.

carriagecontrol is a scalar default CHARACTER variable that evaluates to FORTRAN if the first character of a formatted sequential record is to be 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 can be used as a RECL= specifier in an OPEN statement that connects a file for unformatted direct access when there are input/output statements with the same list of *output-items.*

output-items is a comma-separated list of items used with *iolength* as explained immediately above.

Remarks

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

```
inquire (unit=8, access=acc, err=200)
  ! what access method for unit 8? goto 200 on error
inquire (this_unit, opened=opnd, direct=dir)
  ! is unit this_unit open? direct access allowed?
inquire (file="myfile.dat", recl=record_length)
  ! what is the record length of file "myfile.dat"?
```

INT Function

Description

Convert to INTEGER type.

Syntax

INT (*a*, *kind*)

Required Arguments

a must be of type INTEGER, REAL, or COMPLEX.

Optional Arguments

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

Result

The result is of type INTEGER. If *kind* is present, the kind is that specified by *kind*. The result's value is the value of *a* without its fractional part. If *a* is of type COMPLEX, the result's value is the value of the real part of *a* without its fractional part.

Example

```
b = int(-3.6)  ! b is assigned the value -3
```

INTEGER Statement

Description

The INTEGER statement declares entities of type INTEGER.

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.

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 a data object being declared.

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

Remarks

The same attribute must not appear more than once in a INTEGER statement.

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.

The ALLOCATABLE attribute can be used only when declaring an array that is not a dummy argument or a function result.

An array declared with a POINTER or an ALLOCATABLE attribute must be specified with a deferred shape.

An *array-spec* for a *function-name* that does not have the POINTER attribute must be specified with an explicit shape.

An *array-spec* for a *function-name* that does have the POINTER attribute must be specified with a deferred shape.

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

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

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

The INTENT and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* must 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 attribute if it has the INTRINSIC attribute.

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

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

An *entity* must not be explicitly given any attribute more than once in a scoping unit.

Example

```
integer :: a, b, c    ! a, b, and c are of type integer
integer, dimension (2, 4) :: d
                      !  d is a 2 by 4 array of integers
integer :: e = 2      ! integer e is initialized
```

INTENT Statement

Description

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

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) attribute specifies that the dummy argument is intended to receive data from the invoking scoping unit. The dummy argument must not be redefined or become undefined during the execution of the procedure.

The INTENT (OUT) attribute specifies that the dummy argument is intended to return data to the invoking scoping unit. Any actual argument that becomes associated with such a dummy argument must be definable.

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 scoping unit. Any actual argument that becomes associated with such a dummy argument must be definable.

The INTENT statement must not specify a dummy argument that is a dummy procedure or a dummy pointer.

Example

```
subroutine ex (a, b, c)
  real :: a, b, c
  intent (in) a
  intent (out) b
  intent (in out) c
```

INTERFACE Statement

Description

The INTERFACE statement begins an interface block. An interface block specifies the forms of reference through which a procedure can be invoked. An interface block can be used to specify a procedure interface, a defined operation, or a defined assignment.

Syntax

INTERFACE *[generic-spec]*

Where:

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 *.operator-name*.

operator-name is a user-defined name for the operation, consisting of one to 31 letters.

Remarks

Procedure interface

A procedure interface consists of the characteristics of the procedure, the name of the procedure, the name and characteristics of each dummy argument, and the procedure's generic identifiers, if any.

An interface statement with a *generic-name* specifies a generic interface for each of the procedures in the interface block.

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 functions of two arguments, infix binary operator notation is implied. In the case of functions of one argument, prefix operator notation is implied. 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 dummy data objects and must be specified with INTENT (IN) and 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.).

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 of these subroutines must have exactly two dummy arguments. Each argument must be non-optional. The first argument must have INTENT (OUT) or INTENT (IN OUT) and the second argument must have INTENT (IN). A defined assignment is treated as a reference to the subroutine, with the left-hand side as the first argument and the expession to the right of the equals the second argument. 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

```
interface  ! interface without generic specification
  subroutine ex (a, b, c)
    implicit none
    real, dimension (2,8) :: a, b, c
    intent (in) a
    intent (out) b
  end subroutine ex
  function why (t, f)
    implicit none
    logical, intent (in) :: t, f
    logical :: why
  end function why
end interface

interface swap  ! generic swap routine
  subroutine real_swap(x, y)
    implicit none
    real, intent (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
```

```
interface operator (*)  ! use * for set intersection
  function set_intersection (a, b)
    use set_module  ! contains definition of type set
    implicit none
    type (set), intent (in) :: a, b
    type (set) :: set_intersection
  end function set_intersection
end interface

interface assignment (=)  ! use = for integer to bit
  subroutine integer_to_bit (n, b)
    implicit none
    integer, intent (in) :: n
    logical, intent (out) :: b(:)
  end subroutine integer_to_bit
end interface
```

INTRINSIC Statement

Description

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

Syntax

INTRINSIC *intrinsic-procedure-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

```
intrinsic dlog, dabs  ! dlog and dabs allowed as
                      ! actual arguments
call zee (a, b, dlog)
```

INTRUP Subroutine

Description

Execute a DOS or BIOS function.

Syntax

INTRUP (*intary*, *ntrup*)

Arguments

intary must be a nine-element array of type default INTEGER. It is an INTENT(IN OUT) argument. The elements of the array correspond to the registers EAX, EBX, ECX, EDX, DS, ES, EDI, ESI, and flags, in that order. The registers, except flags, are loaded from the array before the interrupt is executed. All registers, including flags, are assigned back to the array after the interrupt is finished. If the user-supplied selector for DS or ES is not legitimate for the protected-mode environment, then the DS or ES selector that was loaded upon entry to the subroutine will be used. The selector actually used is assigned to the array element corresponding to DS or ES, respectively.

To check whether a particular flag is set after returning from INTRUP, use the following code:

```
if (iand(intary(9), myflag) .NE.  0) then ...
```

where `myflag` is one of the following values:

Table 9: *intary* values

flag	value
carry	1
parity	4
auxiliary carry	16
zero	64
sign	128
trap	256
interrupt enable	512
direction	1024
overflow	2048

ntrup must be of type INTEGER, kind 2. It is an INTENT(IN) argument that is the interrupt number to be executed.

Example

```
call intrup(regs, 21) ! int21 call
```

INVALOP Subroutine

Description

The initial invocation of the INVALOP subroutine masks the invalid operator interrupt on the floating-point unit. *lflag* must be set to true on the first invocation. Subsequent invocations return true or false in the *lflag* variable if the exception has occurred or not occurred, respectively.

Syntax

INVALOP (*lflag*)

Arguments

lflag must be of type LOGICAL. It is assigned the value true if an invalid operation exception has occurred, and false otherwise.

Example

```
call invalop (lflag) ! mask the invalid operation interrupt
```

IOR Function

Description

Bit-wise logical inclusive OR.

Syntax

IOR (*i*, *j*)

Arguments

i must be of type INTEGER.

j must be of type INTEGER and of the same kind as *i*.

Result

The result is of type INTEGER and of the same kind as *i*.

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

Get a runtime I/O error message then continue processing.

Syntax

IOSTAT_MSG (*iostat*, *message*)

Arguments

iostat must be of type INTEGER. It is an INTENT(IN) argument that passes the IOSTAT value from a preceding input/output statement.

message must be of type CHARACTER. It is an INTENT(OUT) argument that is assigned the runtime error message corresponding to the IOSTAT value in *iostat*.

Example

```
call iostat_msg(iostat,msg)  ! msg is assigned
                             ! the runtime error message
                             ! corresponding to iostat
```

ISHFT Function

Description

Bit-wise shift.

Syntax

ISHFT (*i*, *shift*)

Arguments

i must be of type INTEGER.

shift must be of type INTEGER. Its absolute value must be less than the number of bits in *i*.

Result

The result is of type INTEGER and of the same kind as *i*. Its value is the value of *i* shifted by *shift* positions; if *shift* is positive, the shift is to the left, if *shift* is negative, the shift is to the right. Bits shifted out are lost.

Example

```
i = ishft(3,2)  ! i is assigned the value 12
```

ISHFTC Function

Description

Bit-wise circular shift of rightmost bits.

Syntax

ISHFTC (*i*, *shift*, *size*)

Required Arguments

i must be of type INTEGER.

shift must be of type INTEGER. The absolute value of *shift* must be less than or equal to size.

Optional Arguments

size must be of type INTEGER. 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 left by *shift* positions.

Example

```
i = ishftc(13,-2,3)  ! i is assigned the value 11
```

KIND Function

Description

Kind type parameter.

Syntax

KIND (*x*)

Arguments

x can be of any intrinsic type.

Result

The result is a default scalar INTEGER. Its value is equal to the kind type parmater value of *x*.

Example

```
i = kind (0.0) ! i is assigned the value 4
```

LBOUND Function

Description

Lower bounds of an array or a dimension of an array.

Syntax

LBOUND (*array*, *dim*)

Required Arguments

array can be of any type. It must not be a scalar and must not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Arguments

dim must 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 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, dimension (3,-4:0) :: i
integer :: k,j(2)
j = lbound (i)     ! j is assigned the value [1 -4]
k = lbound (i, 2)  ! k is assigned the value -4
```

LEN Function

Description

Length of a CHARACTER data object.

Syntax

LEN (*string*)

Arguments

string must be of type CHARACTER. It can be scalar or array-valued.

Result

The result is a scalar default INTEGER. Its value is the number of characters in *string* or in an element of *string* if *string* is array-valued.

Example

```
i = len ('zee') ! i is assigned the value 3
```

LEN_TRIM Function

Description

Length of a CHARACTER entity without trailing blanks.

Syntax

LEN_TRIM (*string*)

Arguments

string must be of type CHARACTER. It can be scalar or array-valued.

Result

The result is a scalar default INTEGER. Its value is the number of characters in *string* (or in an element of *string* if *string* is array-valued) minus the number of trailing blanks.

Example

```
i = len_trim ('zee  ') ! i is assigned the value 3
i = len_trim ('   ')   ! i is assigned the value zero
```

LGE Function

Description

Test whether a string is lexically greater than or equal to another string based on the ASCII collating sequence.

Syntax

LGE (*string_a*, *string_b*)

Arguments

string_a must be of type default CHARACTER.

string_b must be of type default CHARACTER.

Result

The result is of type default LOGICAL. Its value is true if *string_b* precedes *string_a* in the ASCII collating sequence, or if the strings are the same ignoring trailing blanks; otherwise the result is false. If both strings are of zero length the result is true.

Example

```
l = lge('elephant', 'horse') ! l is assigned the
                             ! value false
```

LGT Function

Description
Test whether a string is lexically greater than another string based on the ASCII collating sequence.

Syntax
LGT (*string_a*, *string_b*)

Arguments
string_a must be of type default CHARACTER.

string_b must be of type default CHARACTER.

Result
The result is of type default LOGICAL. Its value is true if *string_b* precedes *string_a* in the ASCII collating sequence; otherwise the result is false. If both strings are of zero length the result is false.

Example
```
l = lgt('elephant', 'horse') ! l is assigned the
                             ! value false
```

LLE Function

Description
Test whether a string is lexically less than or equal to another string based on the ASCII collating sequence.

Syntax
LLE (*string_a*, *string_b*)

Arguments
string_a must be of type default CHARACTER.

string_b must be of type default CHARACTER.

Result
The result is of type default LOGICAL. Its value is true if *string_a* precedes *string_b* in the ASCII collating sequence, or if the strings are the same ignoring trailing blanks; otherwise the result is false. If both strings are of zero length the result is true.

Example

```
l = lle('elephant', 'horse') ! l is assigned the
                             ! value true
```

LLT Function

Description

Test whether a string is lexically less than another string based on the ASCII collating sequence.

Syntax

LLT (*string_a*, *string_b*)

Arguments

string_a must be of type default CHARACTER.

string_b must be of type default CHARACTER.

Result

The result is of type default LOGICAL. Its value is true if *string_a* precedes *string_b* in the ASCII collating sequence; otherwise the result is false. If both strings are of zero length the result is false.

Example

```
l = llt('elephant', 'horse') ! l is assigned the
                             ! value true
```

LOG Function

Description

Natural logarithm.

Syntax

LOG (*x*)

Arguments

x must be 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*. Its value is equal to a REAL representation of $\log_e x$ if *x* is REAL. Its value is equal to the principal value with imaginary part ω in the range $-\pi < \omega \le \pi$ if *x* is COMPLEX.

Example
```
x = log (3.7) ! x is assigned the value 1.30833
```

LOG10 Function

Description
Common logarithm.

Syntax
LOG10 (*x*)

Arguments
x must be 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 equal to a REAL representation of $\log_{10} x$.

Example
```
x = log10 (3.7) ! x is assigned the value 0.568202
```

LOGICAL Function

Description
Convert between kinds of LOGICAL.

Syntax
LOGICAL (*l*, *kind*)

Required Arguments
l must be of type LOGICAL.

Optional Arguments
kind must be a scalar INTEGER expression that can be evaluated at compile time.

Result

The result is of type LOGICAL. If *kind* is present, the result kind is that of *kind*; otherwise it is of default LOGICAL kind. The result value is that of *l*.

Example

```
l = logical (.true., 4) ! l is assigned the value
                        ! true with kind 4
```

LOGICAL Statement

Description

The LOGICAL statement declares entities of type LOGICAL.

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.

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 a data object being declared.

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

Remarks

The same attribute must not appear more than once in a LOGICAL statement.

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.

The ALLOCATABLE attribute can be used only when declaring an array that is not a dummy argument or a function result.

An array declared with a POINTER or an ALLOCATABLE attribute must be specified with a deferred shape.

An *array-spec* for a *function-name* that does not have the POINTER attribute must be specified with an explicit shape.

An *array-spec* for a *function-name* that does have the POINTER attribute must be specified with a deferred shape.

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

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

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

The INTENT and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* must 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 attribute if it has the INTRINSIC attribute.

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

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

Example

```
logical :: a, b, c    ! a, b, and c are of type logical
logical, dimension (2, 4) :: d
                      ! d is a 2 by 4 array of logical
logical :: e = .true. ! logical e is initialized
```

MATMUL Function

Description

Matrix multiplication.

Syntax

MATMUL (*matrix_a*, *matrix_b*)

Arguments

matrix_a must be of type INTEGER, REAL, COMPLEX, or LOGICAL. It must be array-valued and of rank one or two if *matrix_b* is of rank two, and of rank two if *matrix_b* is of rank one..

matrix_b must be of numerical type if *matrix_a* is of numerical type and of type LOGICAL if *matrix_a* is of type LOGICAL. It must be array-valued and of rank one or two, if *matrix_a* is of rank two, and of rank two if *matrix_a* is of rank one. The size of the first dimension must be the same as the size of the last dimension of *matrix_a*.

Result

If the arguments are of the same numeric type, the result is of that type. If their kinds are the same the result kind is that of the arguments. If their kind is different, the result kind is that of the argument with the greater kind parameter.

If the arguments are of different numeric type and one is of type COMPLEX, then the result is of type COMPLEX. If the arguments are of different numeric type, and neither is of type COMPLEX, the result is of type REAL.

If the arguments are of type LOGICAL, the result is of type LOGICAL. If their kinds are the same the result kind is that of the arguments. If their kind is different, the result kind is that of the argument with the greater kind parameter.

The 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 a(2,3), b(3), c(2)
a = reshape((/1,2,3,4,5,6/), (/2,3/))
                     !  represents  |1 3 5|
                                    |2 4 6|
b = (/1,2,3/)        !  represents  [1,2,3]
c = matmul(a, b)     !  c = [22,28]
```

MAX Function

Description

Maximum value.

Syntax

MAX (*a1*, *a2*, *a3*, ...)

Arguments

The arguments must be of type INTEGER or REAL and must all be of the same type and kind.

Result

The result is of the same type and kind as the arguments. Its value is the value of the largest argument.

Example

```
k = max(-14,3,0,-2,19,1) ! k is assigned the value 19
```

MAXEXPONENT Function

Description

Maximum binary exponent of data type.

Syntax

MAXEXPONENT (*x*)

Arguments

x must be of type REAL. It can be scalar or array-valued.

Result

The result is a scalar default INTEGER. Its value is the largest permissible binary exponent in the data type of *x*.

Example

```
real :: r
integer :: i
i = maxexponent (r)  ! i is assigned the value 128
```

MAXLOC Function

Description

Location of the first element in *array* having the maximum value of the elements identified by *mask*.

Syntax

MAXLOC (*array*, *dim*, *mask*)

Required Arguments

array must be of type INTEGER or REAL. It must not be scalar.

Optional Arguments

dim must be a scalar INTEGER in the range $1 \leq dim \leq n$, where *n* is the rank of *array*. The corresponding dummy argument must not be an optional dummy argument.

mask must be of type LOGICAL and must be conformable with *array*.

Result

The result is of type default INTEGER. If *mask* is absent, the result is a rank one array whose element values are the values of the subscripts of the first element in *array* to have the maximum value of all of the elements of *array*. If *mask* is present, the elements of *array* for which *mask* is false are not considered.

Example

```
integer, dimension(1) :: i
i = maxloc ((/3,0,4,4/)) ! i is assigned the value [3]
```

MAXVAL Function

Description

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 must be of type INTEGER or REAL. It must not be scalar.

Optional Arguments

dim must be a scalar INTEGER in the range $1 \leq dim \leq n$, where *n* is the rank of *array*. The corresponding dummy argument must not be an optional dummy argument.

mask 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 value of the result is the maximum value of all the elements of *array*. If *dim* is present, the value of the result is the maximum value of all elements of *array* along dimension *dim*. If *mask* is present, the elements of *array* for which *mask* is false are not considered.

Example

```
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
! m is the array |1 3|
!                |2 4|
i = maxval(m)          ! i is assigned 4
j = maxval(m,dim=1)    ! j is assigned [2,4]
k = maxval(m,mask=m<3) ! k is assigned 2
```

MERGE Function

Description

Choose alternative values based on the value of a mask.

Syntax

MERGE (*tsource*, *fsource*, *mask*)

Arguments

tsource can be of any type.

fsource must be of the same type and type parameters as *tsource*.

mask must be of type LOGICAL.

Result

The result is of the same type and type parameters as *tsource*. Its value is *tsource* if *mask* is true, and *fsource* otherwise.

Example

```
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
integer, dimension (2,2) :: n = reshape((/3,3,3,3/),(/2,2/))
! m is the array |1 3|
!                |2 4|
! n is the array |3 3|
!                |3 3|
r = merge(m,n,m<n) ! r is assigned (/1,2,3,3/)
```

MIN Function

Description

Minimum value.

Syntax

MIN (*a1*, *a2*, *a3*, ...)

Arguments

The arguments must be of type INTEGER or REAL and must all be of the same type and kind.

Result

The result is of the same type and kind as the arguments. Its value is the value of the smallest argument.

Example

```
k = min(-14,3,0,-2,19,1) ! k is assigned the value -14
```

MINEXPONENT Function

Description

Minimum binary exponent of data type.

Syntax

MINEXPONENT (*x*)

Arguments

x must be of type REAL. It can be scalar or array-valued.

Result

The result is a scalar default INTEGER. Its value is the most negative permissible binary exponent in the data type of *x*.

Example

```
real :: r
integer :: i
i = minexponent (r)  ! i is assigned the value -126
```

MINLOC Function

Description

Location of the first element in *array* having the minimum value of the elements identified by *mask*.

Syntax

MINLOC (*array*, *dim*, *mask*)

Required Arguments

array must be of type INTEGER or REAL. It must not be scalar.

Optional Arguments

dim must be a scalar INTEGER in the range $1 \leq dim \leq n$, where *n* is the rank of *array*. The corresponding dummy argument must not be an optional dummy argument.

mask must be of type LOGICAL and must be conformable with *array*.

Result

The result is of type default INTEGER. If *mask* is absent, the result is a rank one array 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, dimension(1) :: i
i = minloc ((/3,0,4,4/)) ! i is assigned the value [2]
```

MINVAL Function

Description

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 must be of type INTEGER or REAL. It must not be scalar.

Optional Arguments

dim must be a 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 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 value of the result is the minimum value of all the elements of *array*. If *dim* is present, the value of the result is the minimum value of all elements of *array* along dimension *dim*. If *mask* is present, the elements of *array* for which *mask* is false are not considered.

Example

```
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
! m is the array |1 3|
!               |2 4|
i = minval(m)         ! i is assigned 1
j = minval(m,dim=1)   ! j is assigned [1,3]
k = minval(m,mask=m>3) ! k is assigned 4
```

MOD Function

Description

Remainder.

Syntax

MOD (*a*, *p*)

Arguments

a must be of type INTEGER or REAL.

p 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*. Its value is *a* - INT(*a* / *p*) * *p*.

Example

```
r = mod(23.4,4.0) ! r is assigned the value 3.4
i = mod(-23,4)    ! i is assigned the value -3
j = mod(23,-4)    ! j is assigned the value 3
k = mod(-23,-4)   ! k is assigned the value -3
```

MODULE Statement

Description

The MODULE statement begins a module program unit.

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 the same as any local name in the module.

In Lahey Fortran, a module program unit must be compiled before it is used.

Example

```
module m
  implicit none
  type mytype   ! mytype available anywhere m is used
    real :: a, b(2,4)
    integer :: n,o,p
  end type mytype
end module m
subroutine zee ()
  use m
  implicit none
  type (mytype) bee, dee
  ...
end subroutine 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 names
  implicit none
  interface bill
    module procedure fred, jim
  end interface
  contains
  function fred ()
    ...
  end function fred
  function jim ()
    ...
  end function jim
end module names
```

MODULO Function

Description

Modulo.

Syntax

MODULO (*a*, *p*)

Arguments

a must be of type INTEGER or REAL.

p 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 and *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

Copy a sequence of bits from one INTEGER data object to another.

Syntax

MVBITS (*from*, *frompos*, *len*, *to*, *topos*)

Arguments

from must be of type INTEGER. It is an INTENT(IN) argument.

frompos must be of type INTEGER and must be non-negative. It is an INTENT(IN) argument. *frompos* + *len* must be less than or equal to BIT_SIZE(*from*).

len must be of type INTEGER and must be non-negative. It is an INTENT(IN) argument.

to must be a variable of type INTEGER with the same kind as *from*. It can be the same variable as *from*. It is an INTENT(IN OUT) argument. *to* is set by copying *len* bits, starting at position *frompos,* from *from*, to *to*, starting at position *topos*.

topos must be of type INTEGER and must be non-negative. It is an INTENT(IN) argument. *topos* + *len* must be less than or equal to BIT_SIZE(*to*).

Example

```
i = 17; j = 3
call mvbits (i,3,3,j,1) ! j is assigned the value 5
```

NAMELIST Statement

Description

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

Syntax

NAMELIST /*namelist-name*/ *namelist-group [[,] /namelist-name/ namelist-group]* ...

Where:

namelist-name is the name of a namelist group.

namelist-group is a list of variable names.

Remarks

A name in a *namelist-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 *namelist-name* has the public attribute, no item in the *namelist-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

```
namelist /mylist/ x, y, z
```

NBREAK Subroutine

Description

Ignore break interrupts.

Syntax

NBREAK ( )

Remarks

The NBREAK subroutine causes the system to ignore break interrupts (`<Ctrl-C>` or `<Ctrl-Break>`) during execution of the program. If a break is received during console input/output, some data may be lost and an error may result. The error may be trapped using the ERR= or IOSTAT= specifier in the input/output statement.

To return to the system default handling of break interrupts or to capture break interrupts, use the BREAK subroutine (see *"BREAK Subroutine"* beginning on page 75).

Example

```
call nbreak ()     ! ignore break interrupts
```

NDPERR Function

Description

Report floating point exceptions.

Syntax

NDPERR (*lvar*)

Arguments

lvar must be 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

```
exc = ndperr (.true.)
! exc is assigned the bits for floating-point exceptions
! that have occurred.  Exception bits are cleared.
```

NDPEXC Subroutine

Description

Mask all floating point exceptions.

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

Nearest number of a given data type in a given direction.

Syntax

NEAREST (*x*, *s*)

Arguments

x must be of type REAL.

s must be of type REAL and must be non-zero.

Result

The result is of the same type and kind as *x*. Its value is the nearest distinct number, in the data type of *x*, from *x* in the direction of the infinity with the same sign as *s*.

Example

```
a = nearest (34.3, -2.0) ! a is assigned 34.2999954223624
```

NINT Function

Description

Nearest INTEGER.

Syntax

NINT (*a*, *kind*)

Required Arguments

a must be of type REAL.

Optional Arguments

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

Result

The result is of type INTEGER. If *kind* is present the result kind is *kind*; otherwise it is the default INTEGER kind. If $a > 0$, the result has the value

INT(a + 0.5); if $a \le 0$, the result has the value INT(a - 0.5).

Example

```
i = nint (7.73)  ! i is assigned the value 8
i = nint (-4.2)  ! i is assigned the value -4
i = nint (-7.5)  ! i is assigned the value -8
i = nint (2.50)  ! i is assigned the value 3
```

NOT Function

Description

Bit-wise logical complement.

Syntax

NOT (*i*)

Arguments

i must be of type INTEGER.

Result

The result is of the same type and kind as *i*. Its value is the value of *i* with each of its bits complemented (zeros changed to ones and ones changed to zeros).

Example

```
i = not(5) ! i is assigned the value -6
```

NULLIFY Statement

Description

The NULLIFY statement disassociates pointers.

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
```

OFFSET Function

Description

Get the DOS offset portion of the memory address of a variable, substring, array reference, or external subprogram.

Syntax

OFFSET (*item*)

Arguments

item can be of any type. It is the name for which to return an offset. *item* must have the EXTERNAL attribute.

Result

The result is of type INTEGER. It is the offset portion of the memory address of *item*.

Example

```
i = offset(a) ! get the offset portion of the address of a
```

OPEN Statement

Description

The OPEN statement connects or reconnects an external file and 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 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 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, and UNFORMATTED if the file is to be connected for unformatted input/output. The default value is UNFORMATTED, for a file connected for direct access, and FORMATTED, for a file connected for sequential access.

recl is a scalar default INTEGER expression. It must evaluate to the record length for a file connected for direct access, or the maximum record length for a file connected for sequential access.

blank is a scalar default CHARACTER expression. It must evaluate to NULL if null blank control is to be used and ZERO if zero blank control is to be used. The default value is NULL. 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.

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. Sharing modes may also be specified. The are "DENYBOTH" if the file is for exclusive use by this unit in this process; "DENYWRITE" if the file may be read by others, but not written to; "DENYREAD" if the file may be written to by others, but not read; and "DENYNONE" if the file may be read or written to by others. If both access modes (READ, WRITE, or READWRITE) and sharing modes are to be specified, they must be separated by a comma within the same character expression.

delim is a scalar default CHARACTER expression. It must evaluate to APOSTROPHE if the apostrophe will be used to delimit character constants written with list-directed or namelist formatting, QUOTE if the quotation mark will be used, and NONE if neither quotation marks nor apostrophes will be 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.

carriagecontrol is a scalar default CHARACTER expression. It must evaluate to FORTRAN if the first character of a formatted sequential record is to be used for carriage control, and LIST otherwise. Non-storage devices default to FORTRAN; disk files to LIST

Remarks

The OPEN statement can be used to connect an existing file to an input/output unit, create a file that is preconnected, create a file and connect it to an input/output unit, or change certain characteristics of a connection between a file and an input/output unit.

If the 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 already connected to an input/output unit, execution of an OPEN statement on that file and a different unit is not permitted.

FILE= is optional if it is the second argument and the first argument is a unit number with no UNIT=.

Example

```
open (8, file='info.dat',status='new')
```

OPTIONAL Statement

Description

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

Syntax

OPTIONAL *[* :: *] dummy-arg-names*

Where:

dummy-arg-names is a comma-separated list of dummy argument names.

Example

```
subroutine a(b,c)
  real, optional, intent(in) :: c
  ! c need not be provided when calling a
  real, intent(out) :: b
  ...
```

OVEFL Subroutine

Description

The initial invocation of the OVEFL subroutine masks the overflow interrupt on the floating-point unit. *lflag* must be set to true on the first invocation. Subsequent envocations return true or false in the *lflag* variable if the exception has occurred or not occurred, respectively.

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.

Example

```
call ovefl (lflag) ! mask the overflow interrupt
```

PACK Function

Description

Pack an array into a vector under control of a mask.

Syntax

PACK (*array*, *mask*, *vector*)

Required Arguments

array can be of any type. It must not be scalar.

mask must be of type LOGICAL and must be conformable with *array*.

Optional Arguments

vector must be of the same type and kind as *array* and must have rank one. 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, dimension(3,3) :: c
c = reshape((/0,3,2,4,3,2,5,1,2/),(/3,3/))
! represents the array |0 4 5|
!                      |3 3 1|
!                      |2 2 2|
integer, dimension(6) :: d
integer, dimension(9) :: e
d = pack(c,mask=c.ne.2)! d is assigned [0 3 4 3 5 1]
e = pack(c,mask=.true.)! e is assigned [0 3 2 4 3 2 5 1 2]
```

PARAMETER Statement

Description

The PARAMETER statement specifies 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*.

Example

```
parameter (freezing_point = 32.0, conv_factor = 9/5)
```

PAUSE Statement (obsolescent)

Description

The PAUSE statement temporarily suspends execution of the program.

Syntax

PAUSE *[pause-code]*

Where:

pause-code is a scalar CHARACTER constant or a series of 1 to 5 digits.

Remarks

When a PAUSE statement is reached, the optional *pause-code* and the string "`Press enter to continue`" are displayed. The program resumes execution when the `<ENTER>` key is pressed.

Example

```
pause     !"Press enter 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 attribute or the POINTER attribute or a subobject of a variable having the TARGET attribute.

Remarks

If *target* is not a pointer, *pointer* becomes associated with *target*. If *target* is a pointer that is associated, *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.

A pointer also becomes associated with a target through allocation of the pointer.

Any previous association between *pointer* and a target 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
real, target :: b = 5.0
a => b  ! a is an alias for b
```

POINTER Function

Description

Get 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

```
i = pointer(a) ! get the address of a
```

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*)*]] ...*

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*.

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 *variable-name*.

If the DIMENSION attribute is specified elsewhere in the scoping unit, the array must have a deferred shape.

Example

```
real :: next, previous, value
pointer :: next, previous
```

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 must be of type CHARACTER. It is an INTENT(IN) argument whose first character becomes the new precision fill character.

Example

```
call precfill('*')  ! '*' is the new precision fill character
```

PRECISION Function

Description

Decimal precision of data type.

Syntax

PRECISION (*x*)

Arguments

x must be 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

```
i = precision (4.2) ! i is assigned the value 6
```

PRESENT Function

Description

Determine whether an optional argument is present.

Syntax

PRESENT (*a*)

Arguments

a must be an optional 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 and false otherwise.

Example

```
call zee(a, b)
...
subroutine zee (x,y,z)
  implicit none
  real, intent(in out) :: x, y
  real, intent (in), optional :: z
r = present(z) ! r is assigned the value false
```

PRINT Statement

Description

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

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 I*w[.m]*
or B*w[.m]*
or O*w[.m]*
or Z*w[.m]*
or F*w.d*
or E*w.d[*E*e]*
or EN*w.d[*E*e]*
or ES*w.d[*E*e]*
or G*w.d[*E*e]*
or L*w*
or A*[w]*
or D*w.d*

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 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 occurence of the delimiter character.

Example

```
      print*,"hello world"
      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 objects specified 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 must not appear with a list of *access-id*s.

Example

```
module ex
  implicit none
  public     ! default accessibility is public
  real :: a, b, c
  private a  ! a is not accessible outside module
             ! b and c are accessible outside module
  type zee
    private
    integer :: l,m  ! l and m are private
  end type zee
end module ex
```

PRODUCT Function

Description

Product of elements of an array, along a given dimension, for which a mask is true.

Syntax

PRODUCT (*array*, *dim*, *mask*)

Required Arguments

array must be of type INTEGER, REAL or COMPLEX. It must not be scalar.

Optional Arguments

dim must be a 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 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 value of the result is the product of the values of all the elements of *array*. If *dim* is present, the value of the result is the product of the values of all elements of *array* along dimension *dim*. If *mask* is present, the elements of *array* for which *mask* is false are not considered.

Example

```
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
! m is the array |1 3|
!                |2 4|
i = product(m)         ! i is assigned 24
j = product(m,dim=1)   ! j is assigned [2,12]
k = product(m,mask=m>2) ! k is assigned 12
```

PROGRAM Statement

Description

The PROGRAM statement specifies a name for the 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, nor the same as any local name in the main program.

Example

```
program zyx
```

PROMPT Subroutine

Description

Set prompt for subsequent READ statements. Fortran default is no prompt.

Syntax

PROMPT (*message*)

Arguments

message must be of type CHARACTER. It is an INTENT(IN) argument that is the prompt for subsequent READ statements.

Example

```
call prompt('?')  ! ? is the new READ prompt
```

PUBLIC Statement

Description

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

Syntax

PUBLIC *[[::] 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 PUBLIC statement is permitted only in 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 accessibility of the objects specified public.

Example

```
module zee
  implicit none
  private   ! default accessibility is now private
  real :: a, b, c
  public a  ! a is now accessible outside module
end module zee
```

RADIX Function

Description

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*. In Lahey Fortran 90 this value is two for all kinds of INTEGERs and REALs.

Example
```
i = radix(2.3) ! i is assigned the value 2
```

RANDOM_NUMBER Subroutine

Description
Uniformly distributed pseudorandom number or numbers in the range $0 \le x < 1$.

Syntax
RANDOM_NUMBER (*harvest*)

Arguments
harvest must be of type REAL. It is an INTENT(OUT) argument. It can be a scalar or an array variable. Its value is one or several pseudorandom numbers uniformly distributed in the range $0 \le x < 1$.

Example
```
real, dimension(8) :: x
call random_number(x) ! each element of x is assigned
                      ! a pseudorandom number
```

RANDOM_SEED Subroutine

Description
Set or query the pseudorandom number generator used by RANDOM_NUMBER. If no argument is present, the processor sets the seed to a predetermined value.

Syntax
RANDOM_SEED (*size*, *put*, *get*)

Optional Arguments
size must be a scalar of type default INTEGER. It is an INTENT(OUT) variable. It is set to the number of default INTEGERs the processor uses to hold the seed. For Lahey Fortran this value is two.

put must be a default INTEGER array of rank one and size greater than or equal to *size*. It is an INTENT(IN) argument and is used by the processor to set the seed value.

get must be a default INTEGER array of rank one and size greater than or equal to *size*. It is an INTENT(OUT) argument and is set by the processor to the current value of the seed.

Exactly one or zero arguments must be present.

Example

```
call random_seed   ! initialize the generator
call random_seed(size=k)  ! k set to size of seed
call random_seed(put=seed(1:k))  ! set user seed
call random_seed(get=old(1:k))   ! get current seed
```

RANGE Function

Description

Decimal range of the data type of a number.

Syntax

RANGE (*x*)

Arguments

x must be of numeric type.

Result

The result is a scalar default INTEGER. If *x* is of type INTEGER, the result value is INT (LOG10 (*huge*)), where *huge* is the largest positive integer in the data type of *x*. If *x* is of type REAL or COMPLEX, the result value is INT (MIN (LOG10 (*huge*), - LOG10 (*tiny*))), where *huge* and *tiny* are the largest and smallest positive numbers in the data type of *x*.

Example

```
i = range(4.2) ! i is assigned the value 37
```

READ Statement

Description

The READ statement transfers values from an input/output unit to the entities 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 24).

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 exactly one *io-unit*, and must not 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 REC= specifier is present, 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

```
read*, a,b,c  ! read into a, b, and c using list-
              ! directed i/o
read (3, fmt= "(e7.4)") x
              ! read in x from unit 3 using e format
read 10, i,j,k
              ! read in i, j, and k using format at
              ! label 10
```

REAL Function

Description
Convert to REAL type.

Syntax
REAL (*a*, *kind*)

Required Arguments
a must be of type INTEGER, REAL, or COMPLEX.

Optional Arguments
kind must be a scalar INTEGER expression that can be evaluated at compile time.

Result
The result is of type REAL. If *kind* is present, the kind is that specified by *kind*. The result's value is a REAL representation of *a*. If *a* is of type COMPLEX, the result's value is a REAL representation of the real part of *a*.

Example
```
b = real(-3)  ! b is assigned the value -3.0
```

REAL Statement

Description
The REAL statement declares entities of type REAL.

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.

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 a data object being declared.

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

Remarks

The same attribute must not appear more than once in a REAL statement.

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 a blank common, an allocatable array, a pointer, an external name, an intrinsic name, or an automatic object.

The ALLOCATABLE attribute can be used only when declaring an array that is not a dummy argument or a function result.

An array declared with a POINTER or an ALLOCATABLE attribute must be specified with a deferred shape.

An *array-spec* for a *function-name* that does not have the POINTER attribute must be specified with an explicit shape.

An *array-spec* for a *function-name* that does have the POINTER attribute must be specified with a deferred shape.

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

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

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

The INTENT and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* must 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 attribute if it has the INTRINSIC attribute.

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

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

An *entity* must not be explicitly given any attribute more than once in a scoping unit.

Example

```
real :: a, b, c         ! a, b, and c are of type real
real, dimension (2, 4) :: d
                        ! d is a 2 by 4 array of real
real :: e = 2.0         ! real e is initialized
```

REPEAT Function

Description

Concatenate copies of a string.

Syntax

REPEAT (*string*, *ncopies*)

Arguments

string must be scalar and of type CHARACTER

ncopies must be a scalar non-negative INTEGER.

Result

The result is a scalar of type CHARACTER with length equal to *ncopies* times the length of *string*. Its value is equal to the concatenation of *ncopies* copies of *string*.

Example

```
character (len=6) :: n
n = repeat('ho',3) ! n is assigned the value 'hohoho'
```

RESHAPE Function

Description

Construct an array of a specified shape from a given array.

Syntax

RESHAPE (*source*, *shape*, *pad*, *order*)

Required Arguments

source can be of any type and must be array-valued. 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 must be an INTEGER array of rank one and of constant size. Its size must be positive and less than or equal to seven. It must not have any negative elements.

Optional Arguments

pad must be array-valued and of the same type and type parameters as *source*.

order must be of type INTEGER and of the same shape as *shape*. Its value must be a permutation of (1, 2, ..., *n*), where *n* is the size of *shape*. If *order* is absent, it is as if it were present with the value (1, 2, ..., *n*).

Result

The result is an array of shape *shape* with the same type and type parameters as *source*. The elements of the result, taken in permuted subscript order, *order*(1), ..., *order*(*n*), are those of *source* in array element order followed if necessary by elements of one or more copies of *pad* in array element order.

Example

```
x = reshape((/1,2,3,4/), (/3,2/), pad=(/0/))
     ! x is assigned |1 4|
     !               |2 0|
     !               |3 0|
```

RETURN Statement

Description

The RETURN statement completes execution of a procedure and transfers control back to the statement following the procedure invocation.

Syntax

RETURN *[scalar-int-expr]*

Where:

scalar-int-expr is a scalar INTEGER expression.

Remarks

If *scalar-int-expr* is present and has a value *n* between 1 and the number of asterisks in the subprogram's dummy argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate return specifier in the actual argument list.

Example

```
subroutine zee (a, b)
  implicit none
  real, intent(in out) :: a, b
  ...
  if (a>b) then
    return            ! subroutine completed
  else
    a=a*b
    return            ! subroutine completed
  end if
end subroutine zee
```

REWIND Statement

Description

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

Syntax

REWIND *unit-number*

or

REWIND (*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

Rewinding a file that is connected but does not exist has no effect.

Example

```
rewind 10  ! file connected to unit 10 rewound
rewind (10, err = 100)
              ! file connected to unit 10 rewound
              ! on error goto label 100
```

RRSPACING Function

Description

Reciprocal of relative spacing near a given number.

Syntax

RRSPACING (*x*)

Arguments

x must be of type REAL.

Result

The result is of the same type and kind as *x*. Its value is the reciprocal of the spacing, near *x*, of REAL numbers of the kind of *x*.

Example

```
r = rrspacing(-4.7) ! r is assigned the value 0.985662E+07
```

SAVE Statement

Description

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

Syntax

SAVE *[[::] saved-entities]*

Where:

saved-entites 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 is specified in a SAVE statement other than in the main program, it must be specified in every scoping unit in which it appears except in the main program.

A SAVE statement in the main program has no effect.

Example

```
save i,j,/myblock/,k  ! i,j,k and common block
                      ! myblock have the save
                      ! attribute
```

SCALE Function

Description
Multiply a number by a power of two.

Syntax
SCALE (*x*, *i*)

Arguments
x must be of type REAL.

i must be of type INTEGER.

Result
The result is of the same type and kind as *x*. Its value is $x \times 2^i$.

Example
```
x = scale(1.5,3) ! x is assigned the value 12.0
```

SCAN Function

Description
Scan a string for any one of a set of characters.

Syntax
SCAN (*string*, *set*, *back*)

Required Arguments
string must be of type CHARACTER.

set must be of the same kind and type as *string*.

Optional Arguments
back must be of type LOGICAL.

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*.

Example

```
i = scan ("Lalalalala","la") ! i is assigned the
                             ! value 2
i = scan ("LalalaLALA","la",back=.true.)
                             ! i is assigned the
                             ! value 6
```

SEGMENT Function

Description

Get the DOS segment portion of the memory address of a variable, substring, array reference, or external subprogram.

Syntax

SEGMENT (*item*)

Arguments

item can be of any type. It is the name for which to return a segment. *item* must have the EXTERNAL attribute.

Result

The result is of type INTEGER. It is the segment portion of the memory address of *item*.

Example

```
i = segment(a) ! get the segment portion of the address of a
```

SELECT CASE Statement

Description

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.

Syntax

[construct-name : *]* SELECT CASE (*case-expr*)

Where:

construct-name is an optional name for the CASE construct.

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

Remarks

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 be identified by a construct name. If a CASE statement is identified by a *construct-name*, the corresponding SELECT CASE statement must specify the same *construct-name*.

Example

```
select case (i+j)
  case (:-1)
    ...         ! executed if i+j<0
  case (0)
    ...         ! executed if i+j==0
  case (1,4,7)
    ...         ! executed if i+j==(1 or 4 or 7)
  case default
    ...         ! executed if none of the other case
                ! selectors match i+j
end select
```

SELECTED_INT_KIND Function

Description

Kind type parameter of an INTEGER data type that represents all integer values *n* with $-10^r < n < 10^r$.

Syntax

SELECTED_INT_KIND (*r*)

Arguments

r must be a scalar INTEGER.

Result

The result is a scalar of type default INTEGER. Its value is equal to the kind type parameter of the INTEGER data type that accomodates all values *n* with $-10^r < n < 10^r$. If no such kind is available, the result is -1. If more than one kind is available, the return value is the value of the kind type parameter of the kind with the smallest decimal exponent range.

Example

```
integer (kind=selected_int_kind(3)) :: i,j
! i and j are of a data type with a decimal range of
! at least -1000 to 1000
```

SELECTED_REAL_KIND Function

Description

Kind type parameter of a REAL data type with decimal precision of at least *p* digits and a decimal exponent range of at least *r*.

Syntax

SELECTED_REAL_KIND (*p*, *r*)

Optional Arguments

p must be a scalar INTEGER.

r must be a scalar INTEGER.

Result

The result is a scalar of type default INTEGER. Its value is equal to the kind type parameter of the REAL data type with decimal precision of at least *p* digits and a decimal exponent range of at least *r*. If no such kind is available the result is -1 if the precision is not available, -2 if the range is not available, and -3 if neither is available. If more than one kind is available, the return value is the value of the kind type parameter of the kind with the smallest decimal precision.

Example

```
real, (kind=selected_real_kind(3,3)) :: a,b
! a and b are of a data type with a decimal range of
! at least -1000 to 1000 and a precision of at least
! 3 decimal digits
```

SEQUENCE Statement

Description

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

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 for zee
end type zee
```

SET_EXPONENT Function

Description

Model representation of a number with exponent part set to a power of two.

Syntax

SET_EXPONENT (*x*, *i*)

Arguments

x must be of type REAL.

i must be of type INTEGER.

Result

The result is of the same type and kind as *x*. Its value is the FRACTION(*x*)*2^i.

Example

```
a = set_exponent (4.6, 2) ! a is assigned 2.3
```

SHAPE Function

Description

Shape of an array.

Syntax

SHAPE (*source*)

Arguments

source can be of any type and can be array-valued or scalar. It must not be an assumed-size array. It must not be 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*.

Example
```
i = shape(b(1:9,-2:3,10))! i is assigned the value
                         ! (/9,6,10/)
```

SIGN Function

Description
Transfer of sign.

Syntax
SIGN (*a*, *b*)

Arguments
a must be of type INTEGER or REAL.

b must be of the same type and kind as *a*.

Result
The result is of the same type and kind as *a*. Its value is the $|a|$, if *b* is greater than or equal to zero; and $-|a|$, if *b* is less than zero.

Example
```
a = sign (30,-2) ! a is assigned the value -30
```

SIN Function

Description
Sine.

Syntax
SIN (*x*)

Arguments
x must be of type REAL or COMPLEX.

Result

The result is of the same type and kind as *x*. Its value is a REAL or COMPLEX representation of the sine of *x*.

Example

```
r = sin(.5)  ! r is assigned the value 0.479426
```

SINH Function

Description

Hyperbolic sine.

Syntax

SINH (*x*)

Arguments

x must be 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

```
r = sinh(.5)  ! r is assigned the value 0.521095
```

SIZE Function

Description

Size of an array or a dimension of an array.

Syntax

SIZE (*array*, *dim*)

Required Arguments

array can be of any type. It must not be a scalar and must not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Arguments

dim must 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 of type default INTEGER. If *dim* is present, the result is the extent of dimension *dim* of *array*. If *dim* is absent, the result is the number of elements in *array*.

Example

```
integer, dimension (3,-4:0) :: i
integer :: k,j
j = size (i)     ! j is assigned the value 15
k = size (i, 2)  ! k is assigned the value 5
```

SPACING Function

Description

Absolute spacing near a given number.

Syntax

SPACING (*x*)

Arguments

x must be of type REAL.

Result

The result is of the same type and kind as *x*. Its value is the spacing of REAL values, of the kind of *x*, near *x*.

Example

```
x = spacing(4.7) ! x is assigned the value 0.476837E-06
```

SPREAD Function

Description

Adds a dimension to an array by adding copies of a data object along a given dimension.

Syntax

SPREAD (*source*, *dim*, *ncopies*)

Arguments

source can be of any type and can be scalar or array-valued. Its rank must be less than seven.

dim must be a scalar of type INTEGER with a value in the range $1 \le dim \le n + 1$, where n is the rank of *source*.

ncopies must be a scalar of type INTEGER.

Result

The result is an array of the same type and kind as *source* and of rank $n + 1$, where n is the rank of *source*. If *source* is scalar, the shape of the result is MAX(*ncopies*, 0) and each element of the result has a value equal to *source*. If *source* is array-valued with shape (d_1, d_2, ..., d_n), the shape of the result is (d_1, d_2, ..., d_{dim-1}, MAX(*ncopies*, 0), d_{dim-1}, ..., d_n) and the element of the result with subscripts (r_1, r_2, ..., r_{n+1}) has the value *source*(r_1, r_2, ..., r_{dim-1}, r_{dim+1}, ..., r_{n+1}).

Example

```
real, dimension(2) :: b=(/1,2/)
real, dimension(2,3) :: a
a = spread(b,2,3) ! a is assigned |1 1 1|
                                  |2 2 2|
```

SQRT Function

Description

Square Root.

Syntax

SQRT (*x*)

Arguments

x must be of type REAL or COMPLEX. If *x* is REAL, its value must be greater than or equal to zero.

Result

The result is of the same kind and type as *x*. If *x* is of type REAL, the result value is a REAL representation of the square root of *x*. If *x* is of type COMPLEX, the result value is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Example

```
x = sqrt(16.0) ! x is assigned the value 4.0
```

Statement Function Statement

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 terminates execution of the program.

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.

Example

```
if (a>b) then
  stop          ! program execution terminated
end if
```

SUBROUTINE Statement

Description

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

Syntax

[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 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 zee (bar1, bar2)
```

SUM Function

Description

Sum of elements of an array, along a given dimension, for which a mask is true.

Syntax

SUM (*array*, *dim*, *mask*)

Required Arguments

array must be of type INTEGER, REAL, or COMPLEX. It must not be scalar.

Optional Arguments

dim must be a 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 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 value of the result is the sum of the values of all the elements of *array*. If *dim* is present, the value of the result is the sum of the values of all elements of *array* along dimension *dim*. If *mask* is present, the elements of *array* for which *mask* is false are not considered.

Example

```
integer, dimension (2,2) :: m = reshape((/1,2,3,4/),(/2,2/))
! m is the array |1 3|
!                |2 4|
i = sum(m)          ! i is assigned 10
j = sum(m,dim=1)    ! j is assigned [3,7]
k = sum(m,mask=m>2) ! k is assigned 7
```

SYSTEM Subroutine

Description

Execute a DOS command as if from the DOS command line.

Syntax

SYSTEM (*cmd*)

Arguments

cmd must be of type CHARACTER. Its length must not be greater than 122. It is an INTENT(IN) argument that is a DOS command to be executed as if it were typed on the DOS command line. Use of the SYSTEM subroutine for invocation of protected-mode programs is not supported.

Example

```
call system("dir > current.dir")
! puts a listing of the current directory into
! the file 'current.dir'
```

SYSTEM_CLOCK Subroutine

Description

INTEGER data from the real-time clock.

Syntax

SYSTEM_CLOCK (*count*, *count_rate*, *count_max*)

Optional Arguments

count must be a scalar of type default INTEGER. It is an INTENT (OUT) argument. Its value is set to the current value of the processor clock or to

-HUGE(0) if no clock is available.

count_rate must be a scalar of type default INTEGER. It is an INTENT (OUT) argument. It is set to the number of processor clock counts per second, or to zero if there is no clock.

count_max must be a scalar of type default INTEGER. It is an INTENT (OUT) argument. It is set to the maximum value that *count* can have, or zero if there is no clock.

Example

```
call system_clock(c, cr, cm) ! c is set to current
                             ! value of processor
                             ! clock.  cr is set to
                             ! the count_rate, and cm
                             ! is set to the
                             ! count_max
```

TAN Function

Description

Tangent.

Syntax

TAN (*x*)

Arguments

x must be of type REAL.

Result

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

Example

```
r = tan(.5)  ! r is assigned the value 0.546302
```

TANH Function

Description

Hyperbolic tangent.

Syntax

TANH (*x*)

Arguments

x must be 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

```
r = tanh(.5)  ! r is assigned the value 0.462117
```

TARGET Statement

Description

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

Syntax

TARGET *[::] 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

```
target a,b,c  ! a,b, and c have the target attribute
```

TIMER Subroutine

Description

Hundredths of seconds elapsed since midnight.

Syntax

TIMER (*iticks*)

Arguments

iticks must be of type default INTEGER. It is assigned the hundredths of a second elapsed since midnight on the system clock.

Example

```
call timer (iticks)
```

TINY Function

Description

Smallest representable positive number of data type.

Syntax

TINY (*x*)

Arguments

x must be 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

```
a = tiny (4.0) ! a is assigned 0.117549E-37
```

TRANSFER Function

Description

Interpret the physical representation of a number with the type and type parameters of a given number.

Syntax

TRANSFER (*source*, *mold*, *size*)

Required Arguments

source can be of any type.

mold can be of any type.

Optional Arguments

size must be a 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 a scalar and *size* is absent the result is a scalar. If *mold* is array-valued and *size* is absent, the result is array valued and of rank one. Its size is as small as possible such that it is not shorter than *source*. If *size* is present, the result is array-valued of rank one and of size *size*.

If the physical representation of the result is the same length as the physical representation of *source*, the physical representation of the result is that of *source*. If the physical representation of the result is longer than that of source, the physical representation of the leading part of the result is that of *source* and the trailing part is undefined. If the physical representation of the result is shorter than that of source, the physical representation of the result is the leading part of *source.*

Example

```
real :: a
integer :: i
a = transfer(i,a) ! a is assigned the physical
                  ! representation of i
```

TRANSPOSE Function

Description

Transpose an array of rank two.

Syntax

TRANSPOSE (*matrix*)

Arguments

matrix can be of any type. It must be of rank two.

Result

The result is of the same type, kind, and rank as *matrix*. Its shape is (n, m), where (m, n) is the shape of *matrix*. Element (i, j) of the result has the value *matrix*(j, i).

Example

```
integer, dimension(2,3):: a = reshape((/1,2,3,4,5,6/),(/2,3/))
! represents the matrix |1 3 5|
                        |2 4 6|
integer, dimension(3,2) :: b
b = transpose(a) ! b is assigned the value
!                       |1 2|
!                       |3 4|
!                       |5 6|
```

TRIM Function

Description

Omit trailing blanks.

Syntax

TRIM (*string*)

Arguments

string must be of type CHARACTER and must be scalar.

Result

The result is of the same type and kind as *string*. Its value and length are those of *string* with trailing blanks removed.

Example

```
shorter = trim("longer   ")
        ! shorter is assigned the value "longer"
```

Type Declaration Statement

See INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, or TYPE statement.

TYPE Statement

Description

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

Syntax

TYPE (*type-name*) *[*, *attribute-list* :: *] entity [*, *entity]* ...

Where:

type-name is the name of a derived type previously defined in a derived-type definition.

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.

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 a data object being declared.

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

Remarks

The same attribute must not appear more than once in a TYPE statement.

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

The ALLOCATABLE attribute can be used only when declaring an array that is not a dummy argument or a function result.

An array declared with a POINTER or an ALLOCATABLE attribute must be specified with a deferred shape.

An *array-spec* for a *function-name* that does not have the POINTER attribute must be specified with an explicit shape.

An *array-spec* for a *function-name* that does have the POINTER attribute must be specified with a deferred shape.

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

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

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

The INTENT and OPTIONAL attributes can be specified only for dummy arguments.

An *entity* must 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 attribute if it has the INTRINSIC attribute.

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

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

An *entity* must not be given explicitly any attribute more than once in a scoping unit.

Example

```
type zee
  real :: a, b
  integer :: i
end type zee
type (zee) :: a, b, c   ! a, b, and c are of type zee
type (zee), dimension (2, 4) :: d
                        ! d is a 2 by 4 array of type zee
type (zee) :: e = zee(2.0, 3.5, -1)
                        ! e is initialized
```

UBOUND Function

Description

Upper bounds of an array or a dimension of an array.

Syntax

UBOUND (*array*, *dim*)

Required Arguments

array can be of any type. It must not be a scalar and must not be a pointer that is disassociated or an allocatable array that is not allocated.

Optional Arguments

dim must 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)
j = ubound (i)     ! j is assigned the value [3,0]
k = ubound (i, 2)  ! k is assigned the value 0
```

UNDFL Subroutine

Description

The initial invocation of the UNDFL subroutine masks the underflow interrupt on the floating-point unit. *lflag* must be set to true on the first invocation. Subsequent envocations return true or false in the *lflag* variable if the exception has occurred or not occurred, respectively.

Syntax

UNDFL (*lflag*)

Arguments

lflag must be of type LOGICAL. It is assigned the value true if an underflow exception has occurred, and false otherwise.

Example

```
call undfl (lflag) ! mask the underflow interrupt
```

UNPACK Function

Description

Unpack an array of rank one into an array under control of a mask.

Syntax

UNPACK (*vector*, *mask*, *field*)

Arguments

vector can be of any type. It must be of rank one. Its size must be at least as large as the number of true elements in *mask*.

mask must be of type LOGICAL. It must be array-valued.

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./), (/2, 2/) )
e = unpack(c,mask=d,field=-1)
! e is assigned |-1  3|
!               | 0 -1|
```

USE Statement

Description

The USE specifies that a specified module is accessible by the current scoping unit. It also provides a means of renaming or limiting the accessibility of entities in the module.

Syntax

USE *module [, 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 an 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

If no *local-name* is specified, the local name is *use-name*.

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.

An entity can be accessed by more than one *local-name*.

A *local-name* must not be respecified with differing attributes in the scoping unit that contains the USE statement, except that it can appear in a PUBLIC or PRIVATE statement in the scoping unit of a module.

Forward references to modules are not allowed in Lahey Fortran. 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

```
use my_lib, aleph => alpha
    ! use all public entities in my_lib, and
    ! refer to alpha as aleph locally to prevent
    ! conflict with alpha in this_module below
use this_module, only: alpha, beta, operator(+)
    ! use only alpha, beta, and the defined
    ! operator (+) from this_module
```

VAL Function

Description

Pass an item to a procedure by value. VAL can only be used as an actual argument.

Syntax

VAL (*item*)

Arguments

item can be a named data object of type INTEGER, REAL, or LOGICAL. It is the data object for which to return an address. *item* is an INTENT(IN) argument.

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
REAL	8	double
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;};
COMPLEX	8	must not be passed by value; if passed by reference (without CARG) it is a pointer to a structure of the form: struct dp_complex { double real_part; double 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
```

VERIFY Function

Description

Verify that a set of characters contains all the characters in a string.

Syntax

VERIFY (*string*, *set*, *back*)

Required Arguments

string must be of type CHARACTER.

set must be of the same kind and type as *string*.

Optional Arguments

back must be of type LOGICAL.

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.

Example

```
i = verify ("Lalalalala","l") ! i is assigned the
                              ! value 1
i = verify ("LalalaLALA","LA",back=.true.)
                             ! i is assigned the
                             ! value 6
```

WHERE Construct

Description

The WHERE construct controls which elements of an array will be affected by a block of assignment statements. This is also known as masked array assignment.

Syntax

WHERE (*mask-expr*)
    [assignment-stmt]
    [assignment-stmt]
    ...
*[*ELSEWHERE*]*
    [assignment-stmt]
    [assignment-stmt]
    ...
END WHERE

Where:

mask-expr is a LOGICAL expression.

assignment-stmt is an assignment statement.

Remarks

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

```
where (b>c)         ! begin where construct
  b = -1
elsewhere
  b = 1
end where
```

WHERE Statement

Description

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.

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 the 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 24).

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, 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 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 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 REC= specifier is present, 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

```
write (*,*) a,b,c ! write a, b, and c using list-
                  ! directed i/o
write (3, fmt= "(e7.4)") x
                  ! write x to unit 3 using e format
write 10, i,j,k
                  ! write i, j, and k using format on
                  ! line 10
```

YIELD Subroutine

Description

The YIELD subroutine 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.

Syntax

YIELD ( )

Example

```
call yield ()
```

A Fortran 77 Compatibility

This chapter discusses issues that affect the behavior of Fortran 77 code when processed by Lahey Fortran 90.

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. 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

Popular Extensions

In addition to the extensions documented in blue in Chapters 1 and 2, the following popular Fortran 77 extensions are supported for backward compatibility. These features do not provide functionality absent from standard Fortran 90 and they are likely to cause porting problems when moving to other Fortran 90 platforms. Their use in new code is not recommended:

- in fixed source form, if a tab appears in the first six columns, it is replaced by blanks through column 6 if the character following the tab is a letter; otherwise, it is replaced by blanks through column 5 so the character is placed in the continuation character column.
- the ‘$’ character can be used as a non-initial character in a name.
- up to 99 continuation lines are accepted in fixed source form.
- *typespec* * *n* in type declaration statements, e.g., REAL*8, INTEGER*4.
- BYTE as a synonym for INTEGER*1 and DOUBLE COMPLEX as a synonym for COMPLEX*16.
- in a type declaration statement, each item can be initialized by following the name or array declarator with an initial value contained between slashes.
- in certain cases, missing mandatory commas in format specifications are allowed.
- Lahey NAMELIST formatting.

- Lahey E*w.d[*D*e]* edit descriptor.
- the use of the numbers 2 through 9 for carriage control in formatted output.
- a comma in a numeric input field terminates the field regardless of whether the specified width has been exhausted.
- the edit descriptors Q, \, and $.
- the RESULT option may be omitted from scalar recursive functions.
- various intrinsic procedures documented in blue in the appendix *"Intrinsic Procedures."*
- the Lahey RND, RRAND, and RANDS random number routines and DATE and TIME subroutines.

B New in Fortran 90

The following Fortran 90 features were not present in Fortran 77.

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)

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
- elemental intrinsic procedures
- specification of procedure interfaces
- internal procedures

Modules

New Intrinsic Procedures
- 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
  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 59.

All procedures in these tables are intrinsic. VAX/IBM extension procedures, indicated with a dagger, require the *-vax* compiler switch.

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†* *DABS* *IABS* *I2ABS* *IIABS†* *JIABS†*	Numeric REAL_4 REAL_8 REAL_8 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	Numeric COMPLEX_4 COMPLEX_8 REAL_8 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	Absolute Value.	Elemental
AIMAG *DIMAG†*	REAL REAL_8	COMPLEX COMPLEX_8	Imaginary part of a complex number.	Elemental
AINT *DINT*	REAL REAL_8	REAL REAL_8	Truncation to a whole number.	Elemental
ANINT *DNINT*	REAL REAL_8	REAL REAL_8	REAL representation of the nearest whole number.	Elemental
CEILING	INTEGER_4	REAL	Smallest INTEGER greater than or equal to a number.	Elemental
CMPLX *DCMPLX†*	COMPLEX COMPLEX_8	Numeric Numeric	Convert to type COMPLEX.	Elemental
CONJG *DCONJG†*	COMPLEX COMPLEX_8	COMPLEX COMPLEX_8	Conjugate of a complex number.	Elemental
DBLE *DREAL†** *DFLOAT†**	REAL_8 REAL_8 REAL_8	Numeric COMPLEX_8 INTEGER_4	Convert to double-precision REAL type.	Elemental

Table 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
DIM *DDIM* *IDIM* *I2DIM* *IIDIM†* *JIDIM†*	INTEGER or REAL REAL_8 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	INTEGER or REAL REAL_8 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
EXPONENT	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
FRACTION	REAL	REAL	Fraction part of the physical representation of a number.	Elemental
INT *IDINT** *IFIX** *INT2** *INT4** *HFIX†** *IINT†** *JINT†** *IIDINT†** *JIDINT†** *IIFIX†** *JIFIX†**	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_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		
*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		
*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		
MOD	INTEGER or REAL	INTEGER or REAL	Remainder.	Elemental
AMOD	REAL_4	REAL_4		
DMOD	REAL_8	REAL_8		
I2MOD	INTEGER_2	INTEGER_2		
IMOD†	INTEGER_2	INTEGER_2		
JMOD†	INTEGER_4	INTEGER_4		

Table 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
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* *I2NINT* *ININT†* *JNINT†* *IIDNNT†* *JIDNNT†*	INTEGER INTEGER_4 INTEGER_2 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	REAL REAL_8 REAL REAL_4 REAL_4 REAL_8 REAL_8	Nearest INTEGER.	Elemental
REAL *FLOAT** *SNGL** *FLOATI*†* *FLOATJ*†* *DFLOTI*†* *DFLOTJ*†*	REAL REAL_4 REAL_4 REAL_4 REAL_4 REAL_8 REAL_8	Numeric INTEGER REAL_8 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
SCALE	REAL	REAL and INTEGER	Multiply a number by a power of two.	Elemental
SET_EXPONENT	REAL	REAL and INTEGER	Model representation of a number with exponent part set to a power of two.	Elemental

Table 12: Numeric Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
SIGN *DSIGN* *ISIGN* *I2SIGN* *IISIGN†* *JISIGN†*	INTEGER or REAL REAL_8 INTEGER_4 INTEGER_2 INTEGER_2 INTEGER_4	INTEGER or REAL REAL_8 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†* *DCOS*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	Cosine.	Elemental
COSH *DCOSH*	REAL REAL_8	REAL REAL_8	Hyperbolic cosine.	Elemental
EXP *CEXP* *CDEXP†* *DEXP*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	Exponential.	Elemental
LOG *ALOG* *CLOG* *CDLOG†* *DLOG*	REAL or COMPLEX REAL_4 COMPLEX_4 COMPLEX_8 REAL_8	REAL or COMPLEX REAL_4 COMPLEX_4 COMPLEX_8 REAL_8	Natural logarithm.	Elemental
LOG10 *ALOG10* *DLOG10*	REAL REAL_4 REAL_8	REAL REAL_4 REAL_8	Common logarithm.	Elemental

Table 13: Mathematical Functions

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
SIN *CSIN* *CDSIN†* *DSIN*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	Sine.	Elemental
SINH *DSINH*	REAL REAL_8	REAL REAL_8	Hyperbolic sine.	Elemental
SQRT *CSQRT* *CDSQRT†* *DSQRT*	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	REAL or COMPLEX COMPLEX_4 COMPLEX_8 REAL_8	Square root.	Elemental
TAN *DTAN*	REAL REAL_8	REAL REAL_8	Tangent.	Elemental
TANH *DTANH*	REAL REAL_8	REAL REAL_8	Hyperbolic tangent.	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

Table 14: Character Functions

Name	Description	Class
SCAN	Scan a string for any one of a set of characters.	Elemental
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*.	Transformational
MINVAL	Minimum value of elements of an array, along a given dimension, for which a mask is true.	Transformational
PACK	Pack an array into a vector under control of a mask.	Transformational
PRODUCT	Product of elements of an array, along a given dimension, for which a mask is true.	Transformational
RESHAPE	Construct an array of a specified shape from a given array.	Transformational
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.	Transformational
SUM	Sum of elements of an array, along a given dimension, for which a mask is true.	Transformational
TRANSPOSE	Transpose an array of rank two.	Transformational
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.	Transformational

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
MAXEXPO-NENT	Maximum binary exponent of data type.	Inquiry
MINEXPO-NENT	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 $-10^r < n < 10^r$.	Transforma-tional
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*.	Transforma-tional

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_4 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†*	REAL_4 REAL_8	REAL_4 REAL_8	Arccosine in degrees.	Elemental
ALGAMA† *DLGAMA†*	REAL_4 REAL_8	REAL_4 REAL_8	Log gamma function.	Elemental
ASIND† *DASIND†*	REAL_4 REAL_8	REAL_4 REAL_8	Arcsine in degrees.	Elemental
ATAND† *DATAND†*	REAL_4 REAL_8	REAL_4 REAL_8	Arctangent in degrees.	Elemental
ATAN2D† *DATAN2D†*	REAL_4 REAL_8	REAL_4 REAL_8	Arctangent of *y*/*x* (principal value of the argument of the complex number (*x*,*y*)) in degrees.	Elemental
COSD† *DCOSD†*	REAL_4 REAL_8	REAL_4 REAL_8	Cosine in degrees.	Elemental
COTAN† *DCOTAN†*	REAL_4 REAL_8	REAL_4 REAL_8	Contangent.	Elemental
ERF† *DERF†*	REAL_4 REAL_8	REAL_4 REAL_8	Error function.	Elemental
ERFC† *DERFC†*	REAL_4 REAL_8	REAL_4 REAL_8	Error function complement.	Elemental
GAMMA† *DGAMMA†*	REAL_4 REAL_8	REAL_4 REAL_8	Gamma function.	Elemental
SIND† *DSIND†*	REAL_4 REAL_8	REAL_4 REAL_8	Sine in degrees.	Elemental
TAND† *DTAND†*	REAL_4 REAL_8	REAL_4 REAL_8	Tangent in degrees.	Elemental

Table 20: VAX/IBM Intrinsic Functions Without Fortran 90 Equivalents

Name *Specific Names*	**Function Type**	**Argument Type**	**Description**	**Class**
IZEXT†	INTEGER_2	LOGICAL_1	Zero extend.	Elemental
IZEXT2†	INTEGER_2	INTEGER_2		
JZEXT†	INTEGER_4	LOGICAL_4		
JZEXT2†	INTEGER_4	INTEGER_2		
JZEXT4†	INTEGER_4	INTEGER_4		

Table 21: Utility Procedures

Name	Description	Class
BREAK	Handle break interrupts during execution of the program.	Utility Subroutine
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 envocations 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
INTRUP	Execute a DOS or BIOS function.	Utility Subroutine

Table 21: Utility Procedures

Name	Description	Class
INVALOP	The initial invocation of the INVALOP subroutine masks the invalid operator interrupt on the floating-point unit. Subsequent envocations 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
NBREAK	Ignore break interrupts.	Utility Subroutine
NDPERR	Report floating point exceptions.	Utility Function
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 envocations 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

Table 21: Utility Procedures

Name	Description	Class
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 envocations 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 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 of a data object that may be specified in a type declaration statement.

automatic data object: A data object that is a local entity of a subprogram, that is not a dummy argument, and that has a nonconstant CHARACTER length or array bound.

belong: If an EXIT or a CYCLE statement contains a construct name, the statement belongs to the DO construct using that name. Otherwise, it belongs to the innermost DO construct in which it appears.

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: An ordering of all the different characters of a particular kind type parameter.

common block: A block of physical storage that may be accessed by any of the scoping units in 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.

data: Plural of datum.

data entity: A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (called the function result). A data entity has a data type (either intrinsic or derived) and has, or may have, a data value (the exception is an undefined variable). Every data entity has a rank and is thus either a scalar or an array.

data object: A data entity that is a constant, a variable, or a subobject of a constant.

data type: A named category of data that is characterized by a set of values, together with a way to denote these values and a collection of operations that interpret and manipulate the values. For an intrinsic type, the set of data values depends on the values of the type parameters.

datum: A single quantity that may have any of the set of values specified for its data type.

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 nonnegative 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.

initialization expression: An expression that can be evaluated at compile time.

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.

inquiry function: An intrinsic function whose result depends on properties of the principal argument other than the value of the argument.

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.

instance of a subprogram: The copy of a subprogram that is created when a procedure defined by the subprogram is invoked.

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: Statement keyword or argument keyword.

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.

main program: A program unit that is not a module, subprogram, or block data program unit.

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 in FORTRAN 77 that is considered to have been redundant but that is still in frequent use.

operand: An expression that precedes or succeeds an operator.

operation: A computation involving one or two operands.

operator: A lexical token that specifies an operation.

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.

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: For an array, the rank and extents. The shape may be represented by the rank-one array whose elements are the extents in each dimension.

size: For an array, the total number of elements.

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.

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.

E ASCII Character Set

FORTRAN programs may use the full ASCII Character Set as listed below. The characters are listed in collating sequence from first to last. Characters preceded by up arrows (^) are ASCII Control Characters.

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