**intel** ®

# Intel® Fortran Libraries Reference

# Contents

## Chapter 3   POSIX Functions

## Chapter 4   QuickWin Library

**Index**

**Tables**

# *About This Manual*

This manual describes the intrinsic, portability, POSIX, and QuickWin library functions and procedures of the Intel® Fortran libraries. For a description of all the libraries available with Intel Fortran, see Chapter 10, "Libraries" in the *Intel® Fortran Compiler User's Guide*.

This manual is organized as follows:

| | |
|---|---|
| Chapter 1 | Describes the Intel Fortran intrinsic functions |
| Chapter 2 | Describes the portability functions |
| Chapter 3 | Describes the POSIX functions |
| Chapter 4 | Describes QuickWin run-time library functions |

## Related Publications

The following documents provide additional information relevant to the Intel Fortran 95 Language:

The following documents provide additional information relevant to the Intel Fortran Compiler:

- *Fortran 95 Handbook,* Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. The MIT Press, 1997. Provides a comprehensive guide to the standard version of the Fortran 95 Language

- *Fortran 90/95 Explained,* Michael Metcalf and John Reid. Oxford University Press, 1996. Provides a concise description of the Fortran 95 language.

- For Win32-specific information, see the documentation included with the *Microsoft Win32 Software Development Kit*.
- For Microsoft Fortran PowerStation 32 information, see the documentation included with the *Microsoft Fortran Powerstation 32 Development System for Windows NT, Version 1.0*.

Information about the target architecture is available from Intel and from most technical bookstores. Some helpful titles are:

- *Intel® Fortran Programmer's Reference,* doc number 687928
- *Intel® Fortran Compiler User's Guide*, doc number 687931
- *Intel® C/C++ Compiler User's Guide*
- *Intel® Architecture Optimization Reference Manual, Intel Corporation,* doc. number *245127*
- *Intel Processor Identification with the CPUID Instruction*, doc number 241618

Most Intel documents are also available from the Intel Corporation web site at *developer.intel.com*

## Notational Conventions

This manual uses the following conventions:

| | |
|---|---|
| `This type style` | indicates an element of syntax, a reserved word, a keyword, a filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant. |
| `THIS TYPE STYLE` | Fortran source text appears in upper case. |
| | `l` is lowercase letter L in examples. `1` is the number 1 in examples. `O` is the uppercase O in examples. `0` is the number 0 in examples. |
| **`This type style`** | indicates the exact characters you type as input. |
| *`This type style`* | indicates a place holder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the place holder. |
| [*`items`*] | items enclosed in brackets are options. |

| | |
|---|---|
| {*item* \| *item*} | Select only one of the items listed between braces. A vertical bar (\|) separates the items. |
| ... | Ellipses indicate that you can repeat the preceding item. |
| This type style | indicates an Intel Fortran Language extension format. |
| This type style | indicates an Intel Fortran Language extension discussion. Throughout the manual, extensions to the ANSI standard Fortran language appear in this font and color to help you easily identify when your code uses a non-standard language extension. |

# *Intrinsic Procedures*

<div style="text-align: right">1</div>

Intrinsic procedures are built-in functions and subroutines that are available by default to every Fortran 95 program and procedure. (If certain conditions are met, an intrinsic procedure can be made unavailable; see "Availability of Intrinsic Procedures".)

This chapter describes the intrinsic procedures provided by Intel® Fortran. All intrinsic procedures defined by the Fortran 95 Standard are supported in Intel Fortran. In addition, Intel Fortran supports other nonstandard intrinsic procedures to extend the language's functionality; see the section "Nonstandard Intrinsic Procedures". Intel Fortran intrinsic procedures are provided in the library `libintrins.lib`.

The *Intel® Fortran Compiler User's Guide* has more detailed information about the libraries that are shipped with Intel Fortran and the other libraries used by the `ifl` compiler driver, including the `portlib` and `posix` libraries. The `portlib` and `posix` libraries are described in detail in Chapter 2 and Chapter 3 of this manual.

## Overview of Intrinsic Procedures

This section explains the situations under which intrinsic procedures are not available, gives an overview of Intel Fortran intrinsic functions and subroutines, describes the use of the `INTRINSIC` attribute and statement, and discusses nonstandard intrinsic procedures.

## Availability of Intrinsic Procedures

An intrinsic procedure is available in every program unit—except when the intrinsic's name is defined by the user to have a different meaning, such as a representing a user-defined procedure, variable, or constant.

User-defined procedures always take precedence over intrinsic procedures of the same name when the user-defined procedure's definition is visible. This happens, for example, when the user-defined procedure has an explicit interface, is in an EXTERNAL statement, or is a statement function.

Both a user-defined procedure and an intrinsic may have the same name if the user-defined procedure is used to EXTEND a generic intrinsic and the argument types differ.

An intrinsic function is not available when the function's name has been given the EXTERNAL attribute.

## Intrinsic Subroutines and Functions

Intrinsic procedures include both intrinsic functions and intrinsic subroutines.

An intrinsic subroutine is invoked by the CALL statement and can return values through arguments passed to it. An intrinsic function is referenced as part of an expression and upon evaluation returns a value (the "function value"), which is used in the expression.

Intrinsic procedures are identified as either functions or subroutines as part of their "class" identification.

Intrinsic subroutines can be further classified as either elemental or nonelemental subroutines.

Intrinsic functions include the following classes:

- Elemental Functions
- Inquiry Functions
- Transformational Functions

An intrinsic function can be referenced by either a generic name, a specific name, or both; for details see the section "Generic and Specific Intrinsic Function Names".

Each intrinsic's class is noted in the list of intrinsic specifications later in this chapter (see "Intrinsic Procedure Specifications").

## Intrinsic Subroutines

Subroutine references are made by means of the CALL statement. Any values returned by an intrinsic subroutine are provided through the subroutine's arguments.

The sample code segment that follows calls the intrinsic subroutine DATE_AND_TIME to get real-time clock and date data, which DATE_AND_TIME returns by means of the argument Dtime.

```
INTEGER Dtime(8)
CALL DATE_AND_TIME(VALUES=Dtime)
PRINT *, Dtime(1)              ! print the year
```

## Elemental and Nonelemental Subroutines

Intrinsic subroutines include both elemental and nonelemental subroutines.

MVBITS is the only elemental subroutine. All other intrinsic subroutines are nonelemental.

MVBITS is elemental in that it allows arrays to be used as arguments in the same way that scalar arguments are specified. It has all scalar dummy arguments but permits conformable arrays to be passed as actual arguments. The effect is as if the scalar form of the subroutine were called for each corresponding element of the actual argument arrays supplied.

## Intrinsic Functions

A function differs from a subroutine in that a function returns a value, and a function reference is part of an expression (not a complete statement). Each function returns its result as the function value. This value is used in the expression that references the function.

A function reference may occur wherever an expression is allowed. For instance, intrinsic functions may be used in the right-hand side of assignment statements, as arguments to procedures, in output lists, and elsewhere.

In the following segment of code, the SIN intrinsic function is evaluated, then its result is printed:

```
Ar = N*Pi/180     ! angle in radians
PRINT *, SIN(Ar)  ! print sine of angle
```

The statement below assigns the product of Y and the sine of X to the variable Sxy:

```
Sxy = SIN(X) * Y
```

## Generic and Specific Intrinsic Function Names

There are two varieties of intrinsic function names: generic names and specific names. Each intrinsic function has either a generic name, one or more specific names, or both generic and specific names. If both a generic and specific name exist for an intrinsic function, either may be used to invoke it.

The "Generic and Specific Intrinsic Summary" section later in this chapter lists a summary of generic intrinsic functions and their corresponding specific routines (see Table 1-3).

When you reference a generic intrinsic name, the data type of the actual arguments determine which specific intrinsic is invoked. A reference to a specific intrinsic name requires the intrinsic's actual arguments to be of a certain data type.

For instance, the generic intrinsic function ABS can accept arguments of any numeric type. However, a specific version of the intrinsic, DABS, can accept only double precision arguments.

An intrinsic name can be both generic and specific. For example, as shown in Table 1-3, when the intrinsic procedure SIN is called with a double precision argument the specific function DSIN is invoked. When SIN is called with a REAL argument, however, the specific function SIN is invoked.

Using a generic name can, in general, simplify the referencing of intrinsic functions, because the generic name can be specified for multiple types of arguments.

## Elemental Functions

> **NOTE.** *Some command-line options specify different default data type sizes and can cause different or invalid intrinsic procedure references. For details see the section "Data type sizes and command-line options".*

Elemental intrinsic functions allow arrays to be used as arguments in the same way that scalar arguments are specified.

An elemental function that is called with all scalar dummy arguments delivers a scalar result. Calling an elemental function with conformable array arguments, however, results in a conformable array result. The effect is as if a scalar form of the function were called for each corresponding element of the actual argument arrays supplied.

If both array and scalar arguments are specified to an elemental function, each scalar is treated as an array in which all elements have the scalar value. The "scalar array" is conformable with the array arguments.

The following segment of code illustrates how the elemental intrinsic function ABS can be called with both scalar arguments (such as N) and array arguments (such as X).

```
INTEGER N, Nabs
REAL X(5), Xabs(5)
N = -5
Nabs = ABS(N)
X =(/-4.5, 5.2, -3.9, -1.1, 8.7/)
Xabs = ABS(X)
```

After the calls to ABS, *Nabs* has the value 5, and the array Xabs has the value [4.5  5.2  3.9  1.1  8.7].

## Inquiry Functions

Inquiry intrinsic functions return information based on their principal arguments' properties (and not the arguments' values).

The following statements illustrate how the SIZE inquiry function returns information about a property of the array A:

```
REAL A(3:9, 4:10)
INTEGER SizA
SizA = SIZE(A)
```

The SIZE intrinsic function returns either the extent of an array along one dimension or the total number of elements in the array. Because SIZE is an inquiry intrinsic function, only the array's properties, and not the values of the elements of the array, are considered.

Following the call to SIZE, the variable SizA has a value of 49; that is, the total number of elements in array A is 49.

## Transformational Functions

Transformational intrinsic functions include all functions that are not elemental and are not inquiry functions.

In general, transformational functions require at least one array argument, and return either a scalar or array result based on actual arguments that cannot be evaluated elementally. Often, an array result will be of a different shape than the argument(s).

The following code segment makes use of the ANY and ALL transformational intrinsics.

```
INTEGER(4) A(5), B(5)
LOGICAL L1, L2
A =(/3, 5, 7, 9, 9/)
B =(/3, 4, 7, 8, 9/)
L1 = ANY(A .EQ. B)
L2 = ALL(A .EQ. B)
```

The ANY and ALL intrinsics functions determine whether any value or all values are .TRUE. along dimensions of a logical array.

In the above code, the actual argument passed to ANY and ALL is a five-element logical array. This array, which is the result of the expression A .EQ. B, has the value [.TRUE., .FALSE., .TRUE., .FALSE., .TRUE.].

After the above statements are evaluated, the logical variable `L1` has the value `.TRUE.` (at least one element of `A` is equal to the corresponding element of `B`) and `L2` has the value `.FALSE.` (not all elements of `A` are equal to the corresponding elements of `B`).

# INTRINSIC Attribute and Statement

Both the `INTRINSIC` attribute and the `INTRINSIC` statement specify that a name is a specific or generic name of an intrinsic procedure. The `INTRINSIC` and `EXTERNAL` attributes are mutually exclusive.

The `INTRINSIC` attribute and statement typically are used either to document procedures that are intrinsic or to pass intrinsic procedures as actual arguments.

Functions may be declared intrinsic either in an `INTRINSIC` statement or in a type declaration statement using the `INTRINSIC` attribute. Such a declaration may occur only once per name.

Intrinsic subroutine names can be declared intrinsic only by using the `INTRINSIC` statement. The `INTRINSIC` attribute cannot be applied to a subroutine because subroutine names cannot appear in type statements.

## Documenting Intrinsic Procedures

The `INTRINSIC` attribute and statement can be used as documentation techniques to indicate that a name is that of an intrinsic procedure.

This can be useful to other people who will use your code, especially in code where you invoke nonstandard intrinsic procedures.

## Intrinsic Procedures as Actual Arguments

Intrinsic procedures may be passed as actual arguments to subroutines and functions. Only the names of specific intrinsic procedures may be passed, and the compiler must know that the name being passed is that of an intrinsic.

When an intrinsic procedure is used as an actual argument and its name does not appear elsewhere in the same scoping unit, the intrinsic must be declared intrinsic. This can be done with the INTRINSIC attribute (for intrinsic functions) or the INTRINSIC statement (for subroutines and functions).

For example, if the statement

```
CALL My_Subroutine(QSIN)
```

appears in a program unit and no other occurrence of QSIN appears, the compiler assumes that QSIN is a variable and is not the specific name of the intrinsic function SIN. For this reason, QSIN must be declared intrinsic to ensure that the intrinsic function is passed.

> **NOTE.** *Some intrinsic procedures can never be used as actual arguments.*

The following example code shows how the SIN and COS intrinsic functions can be passed to the user-written subroutine My_Subroutine.

```
PROGRAM Example
REAL(4), INTRINSIC :: SIN, COS
CALL My_Subroutine(SIN)
CALL My_Subroutine(COS)
END
SUBROUTINE  My_Subroutine(TrigRtn)
REAL(4), EXTERNAL :: TrigRtn
REAL(4), PARAMETER :: Pi=3.1415926
INTEGER I
DO I=0, 360, 45
     ! Convert degrees to
     ! radians (I*Pi/180) and
     ! call the intrinsic routine
     ! passed as TrigRtn.
   WRITE(6, 100) I," degrees ",TrigRtn(I*Pi/180)
END DO
```

```
100 FORMAT (I4, A9, F12.8)
END
```

Using the `INTRINSIC` attribute, both functions are declared intrinsic in a type declaration statement in the main program unit.

Similarly, in the subroutine, the `EXTERNAL` attribute specifies that the dummy argument `TrigRtn` is the name of a function, not the name of a data object.

## Nonstandard Intrinsic Procedures

In addition to supporting all intrinsic procedures defined by the Fortran 95 Standard, except for `NULL`, Intel Fortran provides additional intrinsic procedures that are nonstandard. The nonstandard intrinsics, like the Standard intrinsics, are part of the Intel Fortran language and are not selected or unselected by command-line options.

For a list of nonstandard intrinsic routine names, see Table 1-3. All intrinsic procedures, including all nonstandard procedures, are described in the section "Intrinsic Procedure Specifications".

The nonstandard intrinsics are included to provide additional functionality not defined in the Standard, to provide compatibility with other Fortran 95 implementations, and to provide specific routines for data types beyond those in the Standard.

> **NOTE.** *Using Intel-supplied nonstandard intrinsic procedures in your code may limit its portability. Other vendors' compliers may not support Intel-supplied nonstandard intrinsics.*

In Intel Fortran, nonstandard intrinsic procedures are supported in the same manner as Standard intrinsics; that is, the routines' generic property, types, and dummy argument attributes are known to the Intel compiler.

The `INTRINSIC` statement allows a non-Intel Fortran compiler, which may not support Intel nonstandard intrinsics, to immediately indicate if the marked procedures are not recognized as an intrinsic routine.

# Data Representation Models

The Fortran 95 Standard specifies data representation models that suggest how data are represented in the computer and how computations are performed on the data. The computations performed by some Fortran 95 intrinsic functions are described in terms of these models.

There are three data representation models in Fortran 95:

- The Bit Model
- The Integer Number System Model
- The Real Number System Model

In a given implementation the model parameters are chosen to match the implementation as closely as possible. However, an exact match is not required and the model does not impose any particular arithmetic on the implementation.

## Data Representation Model Intrinsics

Several intrinsic functions provide information about the three data representation models. These intrinsic are listed in Table 1-1.

**Table 1-1        Intrinsic Functions Related to Data Representation Models**

| Intrinsic function | Description |
|---|---|
| BIT_SIZE(I) | Number of bits in an integer of the kind of I (I is an object, not a kind number) |
| DIGITS(X) | Base digits of precision in integer or real model for X |
| EPSILON(X) | Small value compared to 1 in real model for X |
| EXPONENT(X) | Real model exponent value for X |
| FRACTION(X) | Real model fraction value for X |
| HUGE(X) | Largest model number in integer or real model for X |
| MAXEXPONENT(X) | Maximum exponent value in real model for X |
| MINEXPONENT(X) | Minimum exponent value in real model for X |

**Table 1-1** **Intrinsic Functions Related to Data Representation Models**
(continued)

| Intrinsic function | Description |
| --- | --- |
| NEAREST(X, S) | Nearest processor real value |
| PRECISION(X) | Decimal precision in real model for X |
| RADIX(X) | Base (radix) in integer or real model for X |
| RANGE(X) | Decimal exponent range in integer or real model for X |
| RRSPACING(X) | 1/(relative spacing near X) |
| SCALE(X, I) | X with real model exponent changed by I |
| SET_EXPONENT(X, I) | Set the real model exponent of X to I |
| SPACING(X) | Absolute spacing near X |
| TINY(X) | Smallest number in real model for X |
| DIGITS(X) | Base digits of precision in integer or real model for X |

## The Bit Model

The bit model interprets a nonnegative scalar data object *a* of type integer as a sequence of binary digits (bits), based upon the model

$$a = \sum_{k=0}^{n-1} b_k 2^k$$

where *n* is the number of bits, given by the intrinsic function `BIT_SIZE` and each $b_k$ has a bit value of 0 or 1. The bits are numbered from right to left beginning with 0.

## The Integer Number System Model

The integer number system is modeled by

$$i = s \sum_{k=0}^{q-1} d_k r^k$$

where

| | |
|---|---|
| $i$ | is the integer value |
| $s$ | is the sign (+1 or –1) |
| $r$ | is the radix given by the intrinsic function RADIX (always 2 for Intel® systems) |
| $q$ | is the number of digits (integer greater than 0), given by the intrinsic function DIGITS |
| $dk$ | is the $k$th digit and is an integer $0 \le d_k < r$. The digits are numbered left to right, beginning with 1. |

## The Real Number System Model

The real number system is modeled by

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}$$

where

| | |
|---|---|
| $x$ | is the real value |
| $s$ | is the sign (+1 or –1) |
| $b$ | is the base (real radix) and is an integer greater than 1, given by the intrinsic function RADIX (always 2 for Intel systems) |
| $e$ | is an integer between some minimum value (*lmin*) and maximum value (*lmax*), given by the intrinsic functions MINEXPONENT and MAXEXPONENT |
| $p$ | is the number of mantissa digits and is an integer greater than 1, given by the intrinsic function DIGITS |
| $f_k$ | is the $k$th digit and is an integer $0 \le f_k < b$, but $f_1$ may be zero only if all the $f_k$ are zero. The digits are numbered left to right, beginning with 1. |

# Functional Categories of Intrinsic Procedures

A listing of intrinsic procedures, ordered alphabetically by category, appears in Table 1-2. More complete information on the individual intrinsic procedures is provided in the section "Intrinsic Procedure Specifications".

**Table 1-2    Intrinsic Procedures by Category**

| Category | Intrinsic Routines |
| --- | --- |
| Array construction | MERGE, PACK, SPREAD, UNPACK |
| Array inquiry | ALLOCATED, LBOUND, SHAPE, SIZE, UBOUND |
| Array location | MAXLOC, MINLOC |
| Array manipulation | CSHIFT, EOSHIFT, TRANSPOSE |
| Array reduction | ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM |
| Array reshape | RESHAPE |
| Bit inquiry | BIT_SIZE |
| Bit manipulation | BTEST, IAND, IBCLR, IBITS, IBSET, IEOR, IOR, ISHFT, ISHFTC, MVBITS, NOT |
| Character computation | ACHAR, ADJUSTL, ADJUSTR, CHAR, IACHAR, ICHAR, INDEX, LEN_TRIM, LGE, LGT, LLE, LLT, REPEAT, SCAN, TRIM, VERIFY |
| Character inquiry | LEN |
| Floating-point manipulation | EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, SPACING |
| Kind | KIND, SELECT_INT_KIND, SELECTED_REAL_KIND |
| Logical | LOGICAL |
| Mathematical computation | ACOS, ASIN, ATAN, ATAN2, COS, COSH, EXP, LOG, LOG10, SIN, SINH, SQRT, TAN, TANH |
| Matrix multiply | MATMUL |
| Nonstandard intrinsic procedures | ACOSD, ACOSH, AND, ASIND, ASINH, ATAN2D, ATAND, ATANH, BADDRESS, COSD, DCMPLX, DFLOAT, DNUM, DREAL, HFIX, IACHAR, IADDR, IDIM, IJINT, IMAG, INT1, INT2, INT4, INT8, INUM, ISIGN, ISNAN, IXOR, JNUM, LOC, LSHFT, LSHIFT, MCLOCK, OR,   QNUM, QPROD, RNUM, RSHFT, RSHIFT, SIND, TAND,  XOR |

**Table 1-2     Intrinsic Procedures by Category** (continued)

| Category | Intrinsic Routines |
| --- | --- |
| Numeric computation | ABS, AIMAG, AINT, ANINT, CEILING, CMPLX, CONJG, DBLE, DIM, DPROD, FLOOR, INT, MAX, MIN, MOD, MODULO, NINT, REAL, SIGN |
| Numeric inquiry | DIGITS, EPSILON, HUGE, MAXEXPONENTS, MINEXPONENTS, PRECISION, RADIX, RANGE, TINY |
| Pointer inquiry | ASSOCIATED |
| Prefetching | MM_PREFETCH |
| Presence inquiry | PRESENT |
| Pseudorandom number | RANDOM_NUMBER, RANDOM_SEED |
| Time | DATE_AND_TIME, SYSTEM_CLOCK |
| Transfer | TRANSFER |
| Vector multiply | DOT_PRODUCT |

# Generic and Specific Intrinsic Summary

As mentioned earlier in the section "Generic and Specific Intrinsic Function Names," each intrinsic procedure may have a generic name, one or more specific names, or both generic and specific names. All standard and nonstandard generic and specific intrinsic procedures supported by Intel Fortran are summarized in Table 1-3.

## Summary of Generic and Specific Intrinsic Names

lists a summary of generic and specific intrinsic procedures. The table's listing is alphabetically ordered.

The class information indicates whether an intrinsic is an extension to the Fortran 95 Standard ("nonstandard"). These procedures provide nonstandard behavior. Functions whose names and descriptions appear in this color are also extensions to the Fortran 95 standard.

The class information also indicates which intrinsic procedures are subroutines, which are functions, and whether they are specific or generic. When a number appears in parentheses after a type, such as INTEGER, the number represents the KIND value.

**Table 1-3    Generic and Specific Intrinsic Procedures**

| Intrinsic Procedure | Description |
|---|---|
| `ABS` | Absolute value. |
| | **Class.** generic elemental function |
| | **Summary.** |

```
generic ABS(A)
                INTEGER(1) function BABS(A)
                    INTEGER(1) ::A
                INTEGER(2) function HABS(A)
                    INTEGER(2) ::A
                INTEGER function IABS(A)
                    INTEGER ::A
                INTEGER(8) function ABS(A)
                    INTEGER(8) ::A
                REAL function ABS(A)
                    REAL ::A
                DOUBLE PRECISION function
DABS(A)
                    DOUBLE PRECISION ::A
                REAL function CABS(A)
                    COMPLEX ::A
                DOUBLE PRECISION function
CDABS(A)
                    DOUBLE COMPLEX ::A
                DOUBLE PRECISION function
ZABS(A)
                    DOUBLE COMPLEX ::A
                  REAL(16) function QABS(X)
                    REAL(16) :: X
               COMPLEX(16) function CQABS(X)
                    COMPLEX(16) ::X
                end
```

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| `ACHAR` | Return character corresponding to ASCII value.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic ACHAR(I)`<br>`            CHARACTER function ACHAR(I)`<br>`                INTEGER(1) ::I`<br>`            CHARACTER function ACHAR(I)`<br>`                INTEGER(2) ::I`<br>`            CHARACTER function ACHAR(I)`<br>`                INTEGER(4) ::I`<br>`            CHARACTER function ACHAR(I)`<br>`                INTEGER(8) ::I`<br>`        end` |
| `ACOS` | Arccosine function in radians.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic ACOS(X)`<br>`          REAL function ACOS(X)`<br>`              REAL ::X`<br>`          DOUBLE PRECISION function`<br>`        DACOS(X)`<br>`              DOUBLE PRECISION::X`<br>`        REAL(16) function QACOS(X)`<br>`        QACOS(X)`<br>`              REAL(16) X`<br>`        end` |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| ACOSD | Arccosine function in degrees. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic ACOSD(X)
            REAL function  ACOSD(X)
                REAL(4) ::X
            DOUBLE PRECISION function
        DACOSD(X)
                DOUBLE PRECISION ::X
            REAL(16) function QACOSD(X)
        QACOSD(X)
                REAL(16) X

            end
``` |
| ACOSH | Hyperbolic arccosine of radians. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic ACOSH(X)
            REAL function  ACOSH(X)
                REAL(4) ::X
            DOUBLE PRECISION function
        DACOSH(X)
                DOUBLE PRECISION ::X
            REAL(16) function QACOSH(X)
                REAL(16) ::X

            end
``` |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| ADJUSTL | Adjust string to left, removing leading blanks.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic ADJUSTL(STRING)<br>            CHARACTER function<br>          ADJUSTL(STRING)<br>              CHARACTER ::STRING<br>          end</pre> |
| ADJUSTR | Adjust string to right, removing trailing blanks.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic ADJUSTR(STRING)<br>            CHARACTER function<br>          ADJUSTR(STRING)<br>              CHARACTER ::STRING<br>          end</pre> |
| AIMAG | Imaginary part of a complex number.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic AIMAG(Z)<br>          REAL function AIMAG(Z)<br>              COMPLEX ::Z<br>            DOUBLE PRECISION function<br>          DIMAG(Z)<br>              DOUBLE COMPLEX ::Z<br>          QIMAG(Z)<br>              COMPLEX*32 ::Z<br><br>          end</pre> |
| AIMAX0 | see "MAX(A1, A2, A3, ...)" |
| AIMIN0 | see "MIN(A1, A2, A3, ...)" |

continued

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| AINT | Truncation to a whole number. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | `generic AINT(A,KIND)` |
| | `          REAL(4) function AINT(A,KIND)` |
| | `              REAL(4) ::A` |
| | `              INTEGER,OPTIONAL ::KIND` |
| | `          REAL(8) function DINT(A)` |
| | `              REAL (8)::A` |
| | `           REAL(16) function QINT(A)` |
| | `               REAL(16) ::A` |
| | `      end` |
| AJMAX0 | see "MAX(A1, A2, A3, ...)" |
| AJMIN0 | see "MIN(A1, A2, A3, ...)" |
| AKMAX0 | see "MAX(A1, A2, A3, ...)" |
| AKMIN0 | see "MIN(A1, A2, A3, ...)" |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| `ALL` | Determine whether all values are `.TRUE.` in `MASK` along dimension `DIM`. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | ```
generic ALL(MASK,DIM)
          ! MASK must be array-valued, DIM
          must be scalar
            LOGICAL(1) function
          ALL(MASK,DIM)
                LOGICAL(1) :: MASK
                INTEGER,OPTIONAL::DIM
            LOGICAL(2) function
          ALL(MASK,DIM)
                LOGICAL(2) :: MASK
                INTEGER,OPTIONAL::DIM
            LOGICAL(4) function
          ALL(MASK,DIM)
                LOGICAL(4) :: MASK
                INTEGER,OPTIONAL::DIM
            LOGICAL(8) function
          ALL(MASK,DIM)
                LOGICAL(8) :: MASK
                INTEGER,OPTIONAL::DIM
          end
``` |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| ALLOCATED | Indicate whether an allocated array is currently allocated. |

**Class.** generic inquiry function

**Summary.**

```
generic ALLOCATED(ARRAY)
                ! ARRAY must be an allocatable
                array
                  LOGICAL function
                ALLOCATED(ARRAY)
                      INTEGER(1) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
                      INTEGER(2) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
                      INTEGER(4) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
                      INTEGER(8) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
                      REAL(4) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
                      REAL(8) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
                      COMPLEX(4) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
                      COMPLEX(8) ::ARRAY
                  LOGICAL function
                ALLOCATED(ARRAY)
```

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| ALLOCATED - continued | `LOGICAL(1) ::ARRAY`<br>`  LOGICAL function`<br>`ALLOCATED(ARRAY)`<br>`      LOGICAL(2) ::ARRAY`<br>`  LOGICAL function`<br>`ALLOCATED(ARRAY)`<br>`      LOGICAL(4) ::ARRAY`<br>`  LOGICAL function`<br>`ALLOCATED(ARRAY)`<br>`      LOGICAL(8) ::ARRAY`<br>`  LOGICAL function`<br>`ALLOCATED(ARRAY)`<br>`      CHARACTER ::ARRAY`<br>`  LOGICAL function`<br>`ALLOCATED(ARRAY)`<br>`      `*`DERIVED_TYPE`*` ::ARRAY`<br>`end` |
| `ALOG` | see "LOG(X)" |
| `ALOG10` | see "LOG10(X)" |
| `AMAX0` | see "MAX(A1, A2, A3, ...)" |
| `AMAX1` | see "MAX(A1, A2, A3, ...)" |
| `AMIN0` | see "MIN(A1, A2, A3, ...)" |
| `AMIN1` | see "MIN(A1, A2, A3, ...)" |
| `AMOD` | see "MOD(A, P)" |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| AND | Bitwise AND. |
| | Class. generic elemental nonstandard function |
| | Summary. |
| | generic AND(I,J) |
| |                 INTEGER(1) function AND(I,J) |
| |                     INTEGER(1) ::I,J |
| |                 INTEGER(2) function AND(I,J) |
| |                     INTEGER(2) ::I,J |
| |                 INTEGER(4) function AND(I,J) |
| |                     INTEGER(4) ::I,J |
| |                 INTEGER(8) function AND(I,J) |
| |                     INTEGER(8) ::I,J |
| |              end |
| ANINT | Nearest whole number. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic ANINT(A,KIND) |
| |                 REAL(4) function ANINT(A,KIND) |
| |                     REAL(4) ::A |
| |                     INTEGER,OPTIONAL ::KIND |
| |                 REAL(8) function DNINT(A) |
| |                     REAL(8) ::A |
| |               REAL(16) function QNINT(A,KIND) |
| |                     REAL(16) :: A |
| |              end |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| ANY | Determine whether any value is `.TRUE.` in `MASK` along dimension `DIM`.<br><br>**Class.** generic transformational function<br><br>**Summary.**<br><br>`generic ANY(MASK,DIM)`<br><br>`! MASK must be array-valued, DIM must be scalar`<br>`  LOGICAL(1) function ANY(MASK,DIM)`<br>`    LOGICAL(1) :: MASK; INTEGER,OPTIONAL::DIM`<br>`  LOGICAL(2) function ANY(MASK,DIM)`<br>`    LOGICAL(2) :: MASK; INTEGER,OPTIONAL::DIM`<br>`  LOGICAL(4) function ANY(MASK,DIM)`<br>`    LOGICAL(4) :: MASK; INTEGER,OPTIONAL::DIM`<br>`  LOGICAL(8) function ANY(MASK,DIM)`<br>`    LOGICAL(8) :: MASK; INTEGER,OPTIONAL::DIM`<br>`end` |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| ASIN | Arcsine function in radians. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | `generic ASIN(X)` |
| | `              REAL function   ASIN(X)` |
| | `                  REAL ::X` |
| | `              DASIN(X)` |
| | `                  DOUBLE PRECISION ::X` |
| | `              QASIN(X)` |
| | `                  REAL(16) :: X` |
| | `              end` |
| ASIND | Arcsine function in degrees. |
| | Class. generic elemental nonstandard function |
| | Summary. |
| | `generic ASIND(X)` |
| | `              REAL function ASIND(X)` |
| | `                  REAL ::X` |
| | `              DOUBLE PRECISION function DASIND(X)` |
| | `                  DOUBLE PRECISION ::X` |
| | `              QASIND(X)` |
| | `                  REAL(16) :: X` |
| | `              end` |

**Table 1-3  Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| ASINH | Hyperbolic arcsine of radians.<br>**Class.** generic elemental nonstandard function<br>**Summary.**<br><pre>generic ASINH(X)<br>            REAL function ASINH(X)<br>                REAL ::X<br>            REAL(8) function  DASINH(X)<br>                REAL(8) ::X<br>             REAL(16) function QASINH(X)<br>                REAL(16) :: X<br>        end</pre> |
| ASSOCIATED | Return association status of pointer or indicate if pointer is associated with a target.<br>**Class.** generic inquiry function<br>**Summary.**<br><pre>generic ASSOCIATED(POINTER,TARGET)<br>          ! POINTER must be a pointer.<br>          ! TARGET is optional.<br>          ! TARGET must be a pointer or<br>          target.<br>          ! TARGET may be of any type,<br>          including derived type.<br>            LOGICAL function<br>          ASSOCIATED(POINTER,TARGET)<br>          end</pre> |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| ATAN | Arctangent function in radians. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ``` |
| | generic ATAN(X) |
| |           REAL function ATAN(X) |
| |               REAL ::X |
| |           DOUBLE PRECISION function |
| |       DATAN(X) |
| |               DOUBLE PRECISION ::X |
| |           REAL(16) function QATAN(X) |
| |               REAL(16) :: X |
| |           end |
| ATAN2 | Arctangent function in radians. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic ATAN2(Y,X) |
| |           REAL function ATAN2(Y,X) |
| |               REAL ::Y,X |
| |           DOUBLE PRECISION function |
| |       DATAN2(Y,X) |
| |               DOUBLE PRECISION ::Y,X |
| |        REAL(16) function QATAN2(Y,X) |
| |               REAL(16) :: Y,X |
| |           end |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| ATAN2D | Arctangent function in degrees.<br>**Class.** generic elemental nonstandard function<br>**Summary.** |

```
generic ATAN2D(Y,X)
               REAL function ATAN2D(Y,X)
                   REAL ::Y,X
                 DOUBLE PRECISION function
               DATAN2D(Y,X)
                     DOUBLE PRECISION ::Y,X
               REAL(16) function QATAN2D(Y,X)
                     REAL(16) :: Y,X
               end
```

| ATAND | Arctangent function in degrees.<br>**Class.** generic elemental nonstandard function<br>**Summary.** |
|---|---|

```
generic ATAN2D(Y,X)
               REAL function ATAN2D(Y,X)
                   REAL ::Y,X
                 DOUBLE PRECISION function
               DATAN2D(Y,X)
                     DOUBLE PRECISION ::Y,X
               REAL(16) function QATAN2D(Y,X)
                     REAL(16) :: Y,X
               end
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| ATANH | Hyperbolic arctangent. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic ATANH(X)
              REAL function ATANH(X)
                  REAL ::X
              DOUBLE PRECISION function
          DATANH(X)
                  DOUBLE PRECISION ::X
              REAL(16) function QATANH(X)
                  REAL(16) :: X
          end
``` |
| BABS | see ABS |
| BADDRESS | Return the address of the argument. |
| | Class. generic inquiry nonstandard function |
| | Summary. |
| | INTEGER function BADDRESS(X) |
| | ! X may be of any type, including any derived type. |
| BBCLR | see IBCLR |
| BBITS | see IBITS |
| BBSET | see IBSET |
| BBTEST | see BTEST |
| BDIM | see DIM |
| BIAND | see IAND |
| BIEOR | see IEOR |
| BIOR | see IOR |
| BITEST | see BTEST |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| BIT_SIZE | Return the number of bits in an integer.<br>**Class.** generic inquiry function<br>**Summary.**<br><br>```<br>generic BIT_SIZE(I)<br>          INTEGER(1) function<br>      BIT_SIZE(I)<br>            INTEGER(1) ::I<br>        INTEGER(2) function<br>      BIT_SIZE(I)<br>            INTEGER(2) ::I<br>        INTEGER(4) function<br>      BIT_SIZE(I)<br>            INTEGER(4) ::I<br>        INTEGER(8) function<br>      BIT_SIZE(I)<br>            INTEGER(8) ::I<br>      end<br>``` |
| BIXOR | see IXOR |
| BJTEST | see BTEST |
| BKTEST | see BTEST |
| BMOD | see "MOD(A, P)" |
| BMVBITS | see MVBITS |
| BNOT | see NOT |
| BSHFT | see ISHFT |
| BSHFTC | see BSHFTC |
| BSIGN | see SIGN |

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| BTEST | Bit test of an integer value. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | `generic BTEST(I,POS)` |
| | `LOGICAL(1) function BBTEST(I,POS)` |
| | `INTEGER(1) :: I,POS` |
| | `LOGICAL(2) function BITEST(I,POS)` |
| | `INTEGER(2) :: I,POS` |
| | `LOGICAL(2) function HTEST(I,POS)` |
| | `INTEGER(2) :: I,POS` |
| | `LOGICAL(4) function BJTEST(I,POS)` |
| | `INTEGER(4) :: I,POS` |
| | `LOGICAL(8) function BKTEST(I,POS)` |
| | `INTEGER(8) :: I,POS` |
| | `end` |
| CABS | see `ABS` |
| CCOS | see `COS` |
| CDABS | see `ABS` |
| CDCOS | see `COS` |
| CDEXP | see `EXP` |
| CDLOG | see "LOG(X)" |
| CDSIN | see `SIN` |
| CDSQRT | see `SQRT` |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| CEILING | Return next largest integer.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic CEILING(A)`<br>`            REAL(4) function CEILING(A)`<br>`                REAL(4) ::A`<br>`            REAL(8) function CEILING(A)`<br>`                REAL(8) ::A`<br>`            REAL(16) function CEILING(A)`<br>`                REAL(16) ::A`<br>`        end` |
| CEXP | see EXP |
| CHAR | Return integer corresponding to character value.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic CHAR(I,KIND)`<br>`            INTEGER(1) function`<br>`        CHAR(I,KIND)`<br>`                INTEGER(1) ::I;`<br>`        INTEGER,OPTIONAL ::KIND`<br>`          INTEGER(2) function`<br>`        CHAR(I,KIND)`<br>`                INTEGER(2) ::I;`<br>`        INTEGER,OPTIONAL ::KIND`<br>`          INTEGER(4) function`<br>`        CHAR(I,KIND)`<br>`                INTEGER(4) ::I;`<br>`        INTEGER,OPTIONAL ::KIND`<br>`          INTEGER(8) function`<br>`        CHAR(I,KIND)`<br>`                INTEGER(8) ::I;`<br>`        INTEGER,OPTIONAL ::KIND`<br>`        end` |

<div align="right">continued</div>

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| CLOG | see "LOG(X)" |
| CMPLX | Convert to complex type. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic CMPLX(X,Y,KIND) |
| |            COMPLEX function CMPLX(X,Y,KIND) |
| |                INTEGER(1) ::X; INTEGER(1),OPTIONAL ::Y INTEGER,OPTIONAL ::KIND |
| |            COMPLEX function CMPLX(X,Y,KIND) |
| |                INTEGER(2) ::X; INTEGER(2),OPTIONAL ::Y INTEGER,OPTIONAL ::KIND |
| |            COMPLEX function CMPLX(X,Y,KIND) |
| |                INTEGER(4) ::X; INTEGER(4),OPTIONAL ::Y INTEGER,OPTIONAL ::KIND |
| |            COMPLEX function CMPLX(X,Y,KIND) |
| |                INTEGER(8) ::X; INTEGER(8),OPTIONAL ::Y INTEGER,OPTIONAL ::KIND |
| |            COMPLEX function |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| CMPLX (continued) | ```
CMPLX(X,Y,KIND)
      REAL(4)    ::X;
REAL(4),OPTIONAL ::Y
 INTEGER,OPTIONAL ::KIND
  COMPLEX function
CMPLX(X,Y,KIND)
      REAL(8)    ::X;
REAL(4),OPTIONAL ::Y
      INTEGER,OPTIONAL ::KIND
  COMPLEX function CMPLX(X,KIND)
      COMPLEX(4) ::X;
INTEGER,OPTIONAL ::KIND
  COMPLEX function CMPLX(X,KIND)
      COMPLEX(8) ::X;
INTEGER,OPTIONAL ::KIND
end
``` |
| CONJG | Conjugate of a complex number.<br>**Class.** generic elemental function<br>**Summary.**<br>```
generic CONJG(Z)
      COMPLEX function CONJG(Z)
          COMPLEX ::Z
      DOUBLE COMPLEX function
DCONJG(Z)
          DOUBLE COMPLEX ::Z
      COMPLEX(16) function QCONJG(Z)
          COMPLEX(16) :: Z
end
``` |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| COS | Cosine function in radians. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic COS(X)
        REAL function COS(X)
                REAL ::X
        REAL(8) function DCOS(X)
                DOUBLE PRECISION ::X
        COMPLEX function  CCOS(X)
                COMPLEX ::X
        DOUBLE COMPLEX function CDCOS(X)
                DOUBLE COMPLEX ::X
        DOUBLE COMPLEX function  ZCOS(X)
                DOUBLE COMPLEX ::X
        REAL(16) function QCOS(X)
                REAL(16) :: X
        COMPLEX(16) functon CQCOS(X)
                COMPLEX(16) :: X
        end
``` |
| COSD | Cosine function in degrees. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic COSD(X)
         REAL function COSD(X)
                 REAL ::X
        DOUBLE PRECISION function DCOSD(X)
                DOUBLE PRECISION ::X
            end
``` |

continued

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| COSH | Hyperbolic cosine function.<br>**Class.** generic elemental function<br>**Summary.**<br><br>```<br>generic COSH(X)<br>          REAL function  COSH(X)<br>              REAL ::X<br>           DOUBLE PRECISION function<br>       DCOSH(X)<br>              DOUBLE PRECISION ::X<br>        REAL(16) function QCOSH(X)<br>             REAL(16) :: X<br>        end<br>``` |

<div align="right">continued</div>

**Table 1-3** **Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| COUNT | Count the number of `.TRUE.` elements of `MASK` along dimension `DIM`.<br><br>**Class.** generic transformational function<br><br>**Summary.**<br><pre>generic COUNT(MASK,DIM)<br>            !MASK must be array-valued, DIM<br>            must be scalar<br>              INTEGER function<br>            COUNT(MASK,DIM)<br>                  LOGICAL(1) :: MASK<br>                  INTEGER,OPTIONAL::DIM<br>              INTEGER function<br>            COUNT(MASK,DIM)<br>                  LOGICAL(2) :: MASK<br>                  INTEGER,OPTIONAL::DIM<br>              INTEGER function<br>            COUNT(MASK,DIM)<br>                  LOGICAL(4) :: MASK<br>                  INTEGER,OPTIONAL::DIM<br>              INTEGER function<br>            COUNT(MASK,DIM)<br>                  LOGICAL(8) :: MASK<br>                  INTEGER,OPTIONAL::DIM<br>            end</pre> |
| CPU_TIME | Returns current processor time. To get elapsed CPU_TIME, call the intrinsic twice, once to get the start time, and again to get a finish time, and then subtract start from finish.<br><br>**Class.** generic subroutine<br><br>**Summary.**<br><pre>SUBROUTINE CPU_TIME(TIME)<br>            !TIME must be REAL scalar. Intel<br>            Fortran allows REAL(4), REAL(8),<br>            REAL(16) or DOUBLE-PRECISION</pre> |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| CSHIFT | Circular shift an array expression. |

**Class.** generic transformational function

**Summary.**

```
generic CSHIFT(ARRAY,SHIFT,DIM)
             ! ARRAY must be array-valued,
             ! DIM must be scalar
               INTEGER(1) function
             CSHIFT(ARRAY,SHIFT,DIM)
                   INTEGER(1) ::ARRAY;
             INTEGER :: SHIFT;
             INTEGER,OPTIONAL ::DIM
               INTEGER(2) function
             CSHIFT(ARRAY,SHIFT,DIM)
                   INTEGER(2) ::ARRAY;
             INTEGER :: SHIFT;
             INTEGER,OPTIONAL ::DIM
               INTEGER(4) function
             CSHIFT(ARRAY,SHIFT,DIM)
                   INTEGER(4) ::ARRAY;
             INTEGER :: SHIFT;
             INTEGER,OPTIONAL ::DIM
               INTEGER(8) function
             CSHIFT(ARRAY,SHIFT,DIM)
                   INTEGER(8) ::ARRAY;
              INTEGER :: SHIFT;
            INTEGER,OPTIONAL ::DIM

               REAL(4) function
             CSHIFT(ARRAY,SHIFT,DIM)
                 REAL(4) ::ARRAY;
             INTEGER :: SHIFT;
             INTEGER,OPTIONAL ::DIM
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| CSHIFT (continued) | REAL(8) function CSHIFT(ARRAY,SHIFT,DIM)<br>REAL(8) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>LOGICAL(1) function CSHIFT(ARRAY,SHIFT,DIM)<br>REAL(16) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>LOGICAL(1) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>LOGICAL(2) function CSHIFT(ARRAY,SHIFT,DIM)<br>LOGICAL(2) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>LOGICAL(4) function CSHIFT(ARRAY,SHIFT,DIM)<br>LOGICAL(4) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>LOGICAL(8) function CSHIFT(ARRAY,SHIFT,DIM)<br>LOGICAL(8) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>COMPLEX(4) function CSHIFT(ARRAY,SHIFT,DIM)<br>COMPLEX(4) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| CSHIFT (continued) | COMPLEX(8) function CSHIFT(ARRAY,SHIFT,DIM)<br>COMPLEX(8) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>COMPLEX(16) function CSHIFT(ARRAY,SHIFT,DIM)<br>COMPLEX(16) ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>CHARACTER function CSHIFT(ARRAY,SHIFT,DIM)<br>CHARACTER ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>*DERIVED_TYPE* function CSHIFT(ARRAY,SHIFT,DIM)<br>*DERIVED_TYPE* ::ARRAY; INTEGER :: SHIFT; INTEGER,OPTIONAL ::DIM<br>end |
| CSIN | see SIN |
| CSQRT | see SQRT |
| CTAN | see TAN |
| DABS | see ABS |
| DACOS | see ACOS |
| DACOSD | see ACOSD |
| DACOSH | see ACOSH |
| DASIN | see ASIN |
| DASIND | see ASIND |
| DASINH | see ASINH |
| DATAN | see ATAN |

continued

**Table 1-3** **Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| DATAN2 | see ATAN2 |
| DATAN2D | see ATAN2D |
| DATAND | see ATAND |
| DATANH | see ATANH |
| DATE_AND_TIME | Return current system date and time. |
| | **Class.** generic subroutine |
| | **Summary.** |
| | subroutine |
| | DATE_AND_TIME(DATE,TIME,ZONE,VALUES) |
| | CHARACTER,OPTIONAL ::DATE |
| | CHARACTER,OPTIONAL ::TIME |
| | CHARACTER,OPTIONAL ::ZONE |
| | CHARACTER,OPTIONAL ::VALUES |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| DBLE | Convert to double precision type. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic DBLE(A) |

```
                DOUBLE PRECISION function
            DBLE(A)
                  INTEGER(1) ::A
            DOUBLE PRECISION function
            DBLE(A)
                  INTEGER(2) ::A
            DOUBLE PRECISION function
            DBLE(A)
                  INTEGER(4) ::A
            DOUBLE PRECISION function
            DBLE(A)
                  INTEGER(8) ::A
            DOUBLE PRECISION function
            DBLE(A)
                REAL(4) ::A
            DOUBLE PRECISION function
            DBLE(A)
                REAL(8) ::A
            DOUBLE PRECISION function
            DBLE(A)
                REAL(16) ::A
            DOUBLE PRECISION function
            DBLE(A)
                COMPLEX(4) ::A
            DOUBLE PRECISION function
            DBLE(A)
                COMPLEX(8) ::A
```

continued

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| DBLE (continued) | DOUBLE PRECISION function DBLE(A)<br>        COMPLEX(16) ::A<br>end |
| DBLEQ | see DBLE |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| DCMPLX | Convert to double precision complex type. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |

```
generic DCMPLX(X,Y)
                COMPLEX(8) function
            DCMPLX(X,Y)
                    INTEGER(1) ::X;
            INTEGER(1),OPTIONAL ::Y
              COMPLEX(8) function
            DCMPLX(X,Y)
                    INTEGER(2) ::X;
            INTEGER(2),OPTIONAL ::Y
              COMPLEX(8) function
            DCMPLX(X,Y)
                    INTEGER(4) ::X;
            INTEGER(4),OPTIONAL ::Y
              COMPLEX(8) function
            DCMPLX(X,Y)
                    INTEGER(8) ::X;
            INTEGER(8),OPTIONAL ::Y
            COMPLEX(8) function DCMPLX(X,Y)
                    REAL(4) ::X;
            REAL(4),OPTIONAL ::Y
              COMPLEX(8) function
            DCMPLX(X,Y)
                    REAL(8) ::X;
            REAL(8),OPTIONAL ::Y
              COMPLEX(8) function
            DCMPLX(X,Y)
                    REAL(16) ::X;
            REAL(16),OPTIONAL ::Y
```

continued

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| DCMPLX (continued) | COMPLEX(8) function DCMPLX(X) |
|  | COMPLEX(4) ::X |
|  | COMPLEX(8) function DCMPLX(X) |
|  | COMPLEX(8) ::X |
|  | end |
| DCONJG | see CONJG |
| DCOS | see COS |
| DCOSD | see COSD |
| DCOSH | see COSH |
| DDIM | see DIM |
| DDINT | see AINT |
| DEXP | see EXP |
| DFLOAT | Convert to double precision type. |
|  | **Class.** generic elemental nonstandard function |
|  | **Summary.** |
|  | generic DFLOAT(A) |
|  | DOUBLE PRECISION function DFLOAT(A) |
|  | INTEGER(1) ::A |
|  | DOUBLE PRECISION function DFLOTI(A) |
|  | INTEGER(2) ::A |
|  | DOUBLE PRECISION function DFLOTJ(A) |
|  | INTEGER(4) ::A |
|  | DOUBLE PRECISION function DFLOTK(A) |
|  | INTEGER(8) ::A |
|  | end |
| DFLOTI | see DFLOAT |

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| DFLOTJ | see DFLOAT |
| DFLOTK | see DFLOAT |
| DIGITS | Return number of significant digits in the model. |

**Class.** generic inquiry function

**Summary.**

```
generic DIGITS(X)
                INTEGER function DIGITS(X)
                    INTEGER(1) ::X
                INTEGER function DIGITS(X)
                    INTEGER(2) ::X
                INTEGER function DIGITS(X)
                    INTEGER(4) ::X
                INTEGER function DIGITS(X)
                    INTEGER(8) ::X
                INTEGER function DIGITS(X)
                    REAL(4) ::X
                INTEGER function DIGITS(X)
                    REAL(8) ::X
            end
```

**Table 1-3**     **Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| DIM | Positive difference. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic DIM(X,Y)
            INTEGER(1) function BDIM(X,Y)
                INTEGER(1) ::X,Y
            INTEGER(2) function HDIM(X,Y)
                INTEGER(2) ::X,Y
            INTEGER(4) function IDIM(X,Y)
                INTEGER(4) ::X,Y
            INTEGER(8) function KDIM(X,Y)
                INTEGER(8) ::X,Y
        REAL function DIM(X,Y)
            REAL ::X,Y
        DOUBLE PRECISION function
    DIM(X,Y)
            DOUBLE PRECISION ::X,Y
    end
``` |
| DIMAG | see IMAG |
| DINT | see AINT |
| DLOG | see "LOG(X)" |
| DLOG10 | see "LOG10(X)" |
| DMAX1 | see "MAX(A1, A2, A3, ...)" |
| DMIN1 | see "MIN(A1, A2, A3, ...)" |
| DMOD | see "MOD(A, P)" |
| DNINT | see ANINT |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| DNUM | Convert to double precision. |
| | **Class.** specific elemental nonstandard function |
| | **Summary.** |
| | DOUBLE PRECISION function DNUM(I) |
| | CHARACTER ::I |
| DOT_PRODUCT | Dot product multiplication of numeric or logical vectors. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | generic DOT_PRODUCT(VECTOR_A,VECTOR_B) |
| | Notes. |
| | VECTOR_A must be of numeric type (integer, real, complex) or of logical type. It must be array-valued and of rank one. |
| | VECTOR_B must be of numeric type if VECTOR_A is of numeric type or of logical type if VECTOR_A is of type logical. It must have the same shape as VECTOR_A. |
| | If the arguments are of numeric type: |
| | If VECTOR_A is of type integer or real, the result has value SUM(VECTOR_A*VECTOR_B). |
| | If VECTOR_A is of type complex, the result has value CONJG(VECTOR_A)*VECTOR_B. |
| | If the arguments are of logical type the result has value ANY(VECTOR_A .AND. VECTOR_B). |
| DPROD | Double precision real product. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic DPROD(X,Y) |
| | DOUBLE PRECISION function DPROD(X,Y) |
| | REAL ::X,Y |
| | end |

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| DREAL | Convert to double precision. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |

```
generic DREAL(A)
            DOUBLE PRECISION function
        DREAL(A)
                INTEGER(1) ::A
          DOUBLE PRECISION function
        DREAL(A)
                INTEGER(2) ::A
          DOUBLE PRECISION function
        DREAL(A)
                INTEGER(4) ::A
          DOUBLE PRECISION function
        DREAL(A)
                INTEGER(8) ::A
          DOUBLE PRECISION function
        DREAL(A)
                REAL(4) ::A
          DOUBLE PRECISION function
        DREAL(A)
                REAL(8) ::A
          DOUBLE PRECISION function
        DREAL(A)
                 REAL(16):: A
          DOUBLE PRECISION function
        DREAL(A)
                COMPLEX(4) ::A
          DOUBLE PRECISION function
        DREAL(A)
                COMPLEX(8) ::A
          DOUBLE PRECISION function
         DREAL(A)
                COMPLEX(16) :: A
        end
```

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| DSIGN | see SIGN |
| DSIN | see SIN |
| DSIND | see SIND |
| DSINH | see SINH |
| DSQRT | see SQRT |
| DTAN | see TAN |
| DTAND | see TAND |
| DTANH | see TANH |
| EOSHIFT | End off shift on an array expression. |

EOSHIFT (continued):

**Class.** generic transformational function

**Summary.**

```
generic CSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)
          ! ARRAY must be array-valued
          ! SHIFT must be of type
          integer--see Notes below
          ! DIM must be a scalar integer
          ! BOUNDARY and DIM are optional
            INTEGER(1) function
          EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)
                INTEGER(1) ::ARRAY,
          BOUNDARY
            INTEGER(2) function
          EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)
                INTEGER(2) ::ARRAY,
          BOUNDARY
            INTEGER(4) function
          EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)
                INTEGER(4) ::ARRAY,
          BOUNDARY
            INTEGER(8) function
          EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)
                INTEGER(8) ::ARRAY,
          BOUNDARY
```

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| EOSHIFT (continued) | REAL(4) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)<br>REAL(4) ::ARRAY, BOUNDARY<br>REAL(8) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)<br>REAL(8) ::ARRAY, BOUNDARY<br>LOGICAL(1) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)<br>LOGICAL(1) ::ARRAY, BOUNDARY<br>LOGICAL(2) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)<br>LOGICAL(2) ::ARRAY, BOUNDARY<br>LOGICAL(4) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)<br>LOGICAL(4) ::ARRAY, BOUNDARY<br>LOGICAL(8) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM) |

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| `EOSHIFT (continued)` | ``` LOGICAL(8) ::ARRAY, BOUNDARY   COMPLEX(4) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)     COMPLEX(4) ::ARRAY, BOUNDARY   COMPLEX(8) function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)     COMPLEX(8) ::ARRAY, BOUNDARY   CHARACTER function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)     CHARACTER ::ARRAY, BOUNDARY   DERIVED TYPE function EOSHIFT(ARRAY,SHIFT,BOUNDARY,DIM)     DERIVED_TYPE ::ARRAY, BOUNDARY   end ``` |
| | **Notes**. |
| | `SHIFT` must be of type integer and must be scalar if `ARRAY` has rank one; otherwise see the section "EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)" on page 192. |
| `EPSILON` | Return positive number that is almost negligible compared to unity in the real number model. |
| | **Class.** generic inquiry function |
| | **Summary.** |
| | ``` generic EPSILON(X)         REAL(4) function EPSILON(X)             REAL(4) ::X         REAL(8) function EPSILON(X)             REAL(8) ::X       end ``` |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| EXP | Exponential. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic EXP(X)
                REAL function EXP(X)
                    REAL ::X
                DOUBLE PRECISION function
            DEXP(X)
                    DOUBLE PRECISION ::X
                COMPLEX function CEXP(X)
                    COMPLEX ::X
                DOUBLE COMPLEX function
            CDEXP(X)
                    DOUBLE COMPLEX ::X
                DOUBLE COMPLEX function
            ZEXP(X)
                    DOUBLE COMPLEX ::X
                end
``` |
| EXPONENT | Return the exponent part of the argument when represented as a model number. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic EXPONENT(X)
                INTEGER function EXPONENT(X)
                    REAL(4) ::X
                INTEGER function EXPONENT(X)
                    REAL(8) ::X
                end
``` |
| FLOAT | see REAL |
| FLOATI | see REAL |
| FLOATJ | see REAL |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| FLOATK | see REAL |
| FLOOR | Return the greatest integer less than or equal to the argument.<br><br>**Class.** generic elemental function<br><br>**Summary.**<br><br>```<br>generic FLOOR(A)<br>            REAL(4) function FLOOR(A)<br>                REAL(4) ::A<br>            REAL(8) function FLOOR(A)<br>                REAL(8) ::A<br>            REAL(16) function FLOOR(A)<br>                REAL(16)::A<br>        end<br>``` |
| FRACTION | Return the fractional part of the model representation of the argument value.<br><br>**Class.** generic elemental function<br><br>**Summary.**<br><br>```<br>generic FRACTION(X)<br>            REAL function FRACTION(X)<br>                REAL ::X<br>            DOUBLE PRECISION function<br>        FRACTION(X)<br>                DOUBLE PRECISION ::X<br>          REAL(16) function FRACTION(X)<br>              REAL(16) X<br>        end<br>``` |
| HABS | see ABS |
| HBCLR | see IBCLR |
| HBITS | see IBITS |
| HBSET | see IBSET |
| HDIM | see DIM |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| HFIX | Convert to `INTEGER(2)` type. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | `generic HFIX(A)` |
| | `INTEGER(2) function HFIX(A)` |
| | `INTEGER(1) ::A` |
| | `INTEGER(2) function HFIX(A)` |
| | `INTEGER(2) ::A` |
| | `INTEGER(2) function HFIX(A)` |
| | `INTEGER(4) ::A` |
| | `INTEGER(2) function HFIX(A)` |
| | `INTEGER(8) ::A` |
| | `INTEGER(2) function IIDINT(A)` |
| | `DOUBLE PRECISION ::A` |
| | `INTEGER(2) function HFIX(A)` |
| | `COMPLEX(4) ::A` |
| | `INTEGER(2) function HFIX(A)` |
| | `COMPLEX(8) ::A` |
| | `end` |
| HIAND | see `IAND` |
| HIEOR | see `IEOR` |
| HIOR | see `IOR` |
| HIXOR | see `IXOR` |
| HMOD | see "MOD(A, P)" |
| HMVBITS | see `MVBITS` |
| HNOT | see `NOT` |
| HSHFT | see `ISHFT` |
| HSHFTC | see `ISHFTC` |
| HSIGN | see `SIGN` |
| HTEST | see `BTEST` |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| HUGE | Return the largest number in the model representing numbers. |

**Class.** generic inquiry function

**Summary.**

```
generic HUGE(X)
                INTEGER(1) function HUGE(X)
                    INTEGER(1) ::X
                INTEGER(2) function HUGE(X)
                    INTEGER(2) ::X
                INTEGER(4) function HUGE(X)
                    INTEGER(4) ::X
                INTEGER(8) function HUGE(X)
                    INTEGER(8) ::X
               REAL(4) function HUGE(X)
                   REAL(4) ::X
               REAL(8) function HUGE(X)
                   REAL(8) ::X
             REAL(16) function HUGE(X)
                   REAL(16) HUGE
           end
```

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| IABS | Integer absolute value.<br>**Class.** specific elemental function<br>**Summary.**<br>`generic IABS(A)`<br>`            INTEGER(1) function IABS(A)`<br>`                INTEGER(1) ::A`<br>`            INTEGER(2) function IIABS(A)`<br>`                INTEGER(2) ::A`<br>`            INTEGER(4) function JIABS(A)`<br>`                INTEGER(4) ::A`<br>`            INTEGER(8) function KIABS(A)`<br>`                INTEGER(8) ::A`<br>`        end` |
| IACHAR | Return the position of a character in the ASCII sequence<br>**Class.** generic elemental function<br>**Summary.**<br>`generic IACHAR(C)`<br>`            INTEGER function IACHAR(C)`<br>`                CHARACTER ::C`<br>`        end` |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| IADDR | Return the address of the argument. |
| | **Class.** specific inquiry nonstandard function |
| | **Summary.** |
| | INTEGER function IADDR(X) |
| | ! X may be any one of the following: |
| | INTEGER(1) ::X |
| | INTEGER(2) ::X |
| | INTEGER(4) ::X |
| | INTEGER(8) ::X |
| | REAL(4) ::X |
| | REAL(8) ::X |
| | COMPLEX(4) ::X |
| | COMPLEX(8) ::X |
| | LOGICAL(1) ::X |
| | LOGICAL(2) ::X |
| | LOGICAL(4) ::X |
| | LOGICAL(8) ::X |
| | CHARACTER ::X |
| | *DERIVED_TYPE* :: X |

continued

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| IAND | Bitwise logical AND. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | `generic IAND(I,J)` |
| | `              INTEGER(1) function BIAND(I,J)` |
| | `                  INTEGER(1) ::I,J` |
| | `              INTEGER(2) function HIAND(I,J)` |
| | `                  INTEGER(2) ::I,J` |
| | `              INTEGER(2) function IIAND(I,J)` |
| | `                  INTEGER(2) ::I,J` |
| | `              INTEGER(4) function JIAND(I,J)` |
| | `                  INTEGER(4) ::I,J` |
| | `              INTEGER(8) function KIAND(I,J)` |
| | `                  INTEGER(8) ::I,J` |
| | `          end` |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| IBCLR | Clear a bit to zero.<br><br>**Class.** generic elemental function<br><br>**Summary.**<br><br><pre>generic IBCLR(I,POS)<br>            INTEGER(1) function<br>        BBCLR(I,POS)<br>                INTEGER(1) :: I,POS<br>          INTEGER(2) function<br>        HBCLR(I,POS)<br>                INTEGER(2) :: I,POS<br>          INTEGER(2) function<br>        IIBCLR(I,POS)<br>                INTEGER(2) :: I,POS<br>          INTEGER(4) function<br>        JIBCLR(I,POS)<br>                INTEGER(4) :: I,POS<br>          INTEGER(8) function<br>        KIBCLR(I,POS)<br>                INTEGER(8) :: I,POS<br>        end</pre> |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| IBITS | Extract a sequence of bits. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic IBITS(I,POS,LEN) |

```
generic IBITS(I,POS,LEN)
                INTEGER(1) function
              BBITS(I,POS,LEN)
                    INTEGER(1) :: I,POS,LEN
              INTEGER(2) function
              HBITS(I,POS,LEN)
                    INTEGER(2) :: I,POS,LEN
              INTEGER(2) function
              IIBITS(I,POS,LEN)
                    INTEGER(2) :: I,POS,LEN
              INTEGER(4) function
              JIBITS(I,POS,LEN)
                    INTEGER(4) :: I,POS,LEN
              INTEGER(8) function
              KIBITS(I,POS,LEN)
                    INTEGER(8) :: I,POS,LEN
              end
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| IBSET | Set a bit to one.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic IBSET(I,POS)<br>              INTEGER(1) function<br>           BBSET(I,POS)<br>                 INTEGER(1) :: I,POS<br>             INTEGER(2) function<br>           HBSET(I,POS)<br>                 INTEGER(2) :: I,POS<br>             INTEGER(2) function<br>           IIBSET(I,POS)<br>                 INTEGER(2) :: I,POS<br>             INTEGER(4) function<br>           JIBSET(I,POS)<br>                 INTEGER(4) :: I,POS<br>             INTEGER(8) function<br>           KIBSET(I,POS)<br>                 INTEGER(8) :: I,POS<br>           end</pre> |
| ICHAR | Return ASCII value corresponding to character.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic ICHAR(C)<br>           INTEGER function ICHAR(C)<br>               CHARACTER ::C<br>           end</pre> |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| IDIM | Integer positive difference. |
| | **Class.** specific elemental nonstandard function |
| | **Summary.** |
| | ```
generic function DIM(X,Y)
                INTEGER(1) function BDIM(X,Y)
                    INTEGER(1) ::X,Y
                INTEGER(2) function IIDIM(X,Y)
                    INTEGER(2) ::X,Y
                INTEGER(4) function JIDIM(X,Y)
                    INTEGER(4) ::X,Y
                INTEGER(8) function KIDIM(X,Y)
                    INTEGER(8) ::X,Y
            end
``` |
| IDINT | see INT8 |
| IDNINT | see NINT |
| IEOR | Bitwise exclusive OR. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic IEOR(I,J)
                INTEGER(1) function BIEOR(I,J)
                    INTEGER(1) ::I,J
                INTEGER(2) function HIEOR(I,J)
                    INTEGER(2) ::I,J
                INTEGER(2) function IIEOR(I,J)
                    INTEGER(2) ::I,J
                INTEGER(4) function JIEOR(I,J)
                    INTEGER(4) ::I,J
                INTEGER(8) function KIEOR(I,J)
                    INTEGER(8) ::I,J
            end
``` |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| IFIX | see INT |
| IIABS | see IABS |
| IIAND | see IAND |
| IIBCLR | see IBCLR |
| IIBITS | see IBITS |
| IIBSET | see IBSET |
| IIDIM | see IDIM |
| IIDINT | see IDINT |
| IIDNNT | see NINT |
| IIEOR | see IEOR |
| IIFIX | see INT |
| IINT | see INT |
| IIOR | see IOR |
| IIQINT | see IQINT |
| IIQNNT | see NINT |
| IISHFT | see ISHFT |
| IISHFTC | see ISHFTC |
| IISIGN | see SIGN |
| IIXOR | see IXOR |
| IJINT | Convert to INTEGER(2). |
| | **Class.** specific elemental nonstandard function |
| | **Summary.** |
| | INTEGER(2) function IJINT(A) |
| | INTEGER(4) ::A |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| IMAG | Imaginary part of a complex number.<br><br>**Class.** generic elemental nonstandard function<br><br>**Summary.**<br><br>`generic IMAG(Z)`<br><br>`REAL function IMAG(Z)`<br>`COMPLEX ::Z`<br>`DOUBLE PRECISION function DIMAG(Z)`<br>`DOUBLE COMPLEX ::Z`<br>`end` |
| IMAX0 | see "MAX(A1, A2, A3, ...)" |
| IMAX1 | see "MAX(A1, A2, A3, ...)" |
| IMIN0 | see "MIN(A1, A2, A3, ...)" |
| IMIN1 | see "MIN(A1, A2, A3, ...)" |
| IMOD | see "MOD(A, P)" |
| INDEX | Return the starting position of a substring within a string.<br><br>**Class.** generic elemental function<br><br>**Summary.**<br><br>`generic INDEX(STRING,SUBSTRING,BACK)`<br><br>`INTEGER function INDEX(STRING,SUBSTRING,BACK)`<br>`CHARACTER::STRING`<br>`CHARACTER::SUBSTRING`<br>`LOGICAL,OPTIONAL::BACK`<br>`end` |
| ININT | see NINT |
| INOT | see NOT |

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| INT | Convert to integer type.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic INT(A,KIND)`<br>       `INTEGER(2) function IINT(A)`<br>           `REAL(4) ::A`<br>     `INTEGER function INT(A, KIND)`<br>        `INTEGER(1) ::A;`<br>        `INTEGER,OPTIONAL ::KIND`<br>     `INTEGER function INT(A, KIND)`<br>        `INTEGER(2) ::A`<br>        `INTEGER,OPTIONAL ::KIND`<br>     `INTEGER(4) function IFIX(A)`<br>        `REAL ::A`<br>    `INTEGER(8) function KIFIX(A)`<br>        `REAL(4)  ::A`<br>    `INTEGER(8) function KINT(A`<br>        `REAL(4) ::A`<br>    `INTEGER(4) function JIFIX(A)`<br>        `REAL(4) ::A`<br>     `INTEGER(4) function JINT(A)`<br>        `REAL(4) ::A`<br>    `INTEGER(4) function IDINT(A)`<br>        `DOUBLE PRECISION ::A`<br>    `INTEGER(4) function JIDINT(A)`<br>        `DOUBLE PRECISION ::A`<br>    `INTEGER(2) function IIQINT(A)`<br>        `REAL(16) :: A`<br>     `INTEGER function INT(A, KIND)`<br>        `COMPLEX(4) ::A` |

<div align="right">continued</div>

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| INT (continued) | `                    INTEGER,OPTIONAL ::KIND`<br>`INTEGER function INT(A, KIND)`<br>`      COMPLEX(8) ::A`<br>`      INTEGER,OPTIONAL ::KIND`<br>`INTEGER function INT(A,KIND)`<br>`      COMPLEX(16) :: A`<br>`      INTEGER,OPTIONAL ::KIND`<br>`end` |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| INT1 | Convert to INTEGER(1) type. |

**Class.** generic elemental nonstandard function

**Summary.**

```
generic INT1(A)
                    INTEGER(1) function INT1(A)
                        INTEGER(1) ::A
                    INTEGER(1) function INT1(A)
                        INTEGER(2) ::A
                    INTEGER(1) function INT1(A)
                        INTEGER(4) ::A
                    INTEGER(1) function INT1(A)
                        INTEGER(8) ::A
                    INTEGER(1) function INT1(A)
                        REAL(4) ::A
                    INTEGER(1) function INT1(A)
                        REAL(8) ::A
                    INTEGER(1) function INT1(A)
                        REAL(16) :: A
                    INTEGER(1) function INT1(A)
                        COMPLEX(4) ::A
                    INTEGER(1) function INT1(A)
                        COMPLEX(8) ::A
                    INTEGER(1) function INT1(A)
                        COMPLEX(16) :: A
                    INTEGER(1) function INT1(A)
                        LOGICAL(1) ::A
                    INTEGER(1) function INT1(A)
                        LOGICAL(2) ::A
                    INTEGER(1) function INT1(A)
                        LOGICAL(4) ::A
                    INTEGER(1) function INT1(A)
                        LOGICAL(8) ::A
                end
```

continued

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| INT2 | Convert to INTEGER(2) type. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | generic INT2(A) |

```
                        INTEGER(2) function INT2(A)
                            INTEGER(1) ::A
                        INTEGER(2) function INT2(A)
                            INTEGER(2) ::A
                        INTEGER(2) function INT2(A)
                            INTEGER(4) ::A
                        INTEGER(2) function INT2(A)
                            INTEGER(8) ::A
                        INTEGER(2) function INT2(A)
                            REAL(4) ::A
                        INTEGER(2) function INT2(A)
                            REAL(8) ::A
                        INTEGER(2) function INT2(A)
                            COMPLEX(4) ::A
                        INTEGER(2) function INT2(A)
                            COMPLEX(8) ::A
                        INTEGER(2) function INT2(A)
                            LOGICAL(1) ::A
                        INTEGER(2) function INT2(A)
                            LOGICAL(2) ::A
                        INTEGER(2) function INT2(A)
                            LOGICAL(4) ::A
                        INTEGER(2) function INT2(A)
                            LOGICAL(8) ::A
                    end
```

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| INT4 | Convert to INTEGER(4) type. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | generic INT4(A) |

```
                  INTEGER(4) function INT4(A)
                      INTEGER(1) ::A
                  INTEGER(4) function INT4(A)
                      INTEGER(2) ::A
                  INTEGER(4) function INT4(A)
                      INTEGER(4) ::A
                  INTEGER(4) function INT4(A)
                      INTEGER(8) ::A
                  INTEGER(4) function INT4(A)
                      REAL(4) ::A
                  INTEGER(4) function INT4(A)
                      REAL(8) ::A
                  INTEGER(4) function INT4(A)
                      REAL(16) :: A
                  INTEGER(4) function INT4(A)
                      COMPLEX(4) ::A
                  INTEGER(4) function INT4(A)
                      COMPLEX(8) ::A
                  INTEGER(4) function INT4(A)
                      COMPLEX(16) :: A
              end
```

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| INT8 | Convert to INTEGER(8) type. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | generic INT8(A) |

```
                      INTEGER(8) function INT8(A)
                          INTEGER(1) ::A
                      INTEGER(8) function INT8(A)
                          INTEGER(2) ::A
                      INTEGER(8) function INT8(A)
                          INTEGER(4) ::A
                      INTEGER(8) function INT8(A)
                          INTEGER(8) ::A
                      INTEGER(8) function INT8(A)
                          REAL(4) ::A
                      INTEGER(8) function INT8(A)
                          REAL(8) ::A
                      INTEGER(8) function INT8(A)
                          REAL(16) ::A
                      INTEGER(8) function INT8(A)
                          COMPLEX(4) ::A
                      INTEGER(8) function INT8(A)
                          COMPLEX(8) ::A
                     INTEGER(8) function KIDINT(A)
                          DOUBLE PRECISION ::A
                     INTEGER(8) function INT8(A)
                          COMPLEX(16) :: A
                    INTEGER(8) function KIDINT(A)
                          DOUBLE PRECISION ::A
                  end
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| INUM | Convert character to INTEGER(2) type. |
| | **Class.** specific elemental nonstandard function |
| | **Summary.** |
| | INTEGER(2) function INUM(I) |
| | CHARACTER ::INOTE: The ASCII characters in the CHARACTER expression must be numeric characters, and the result must fit in an INTEGER(2) number. |
| IOR | Logical OR. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic IOR(I,J) |
| | INTEGER(1) function BIOR(I,J) |
| | INTEGER(1) ::I,J |
| | INTEGER(2) function HIOR(I,J) |
| | INTEGER(2) ::I,J |
| | INTEGER(2) function IIOR(I,J) |
| | INTEGER(2) ::I,J |
| | INTEGER(4) function JIOR(I,J) |
| | INTEGER(4) ::I,J |
| | INTEGER(8) function KIOR(I,J) |
| | INTEGER(8) ::I,J |
| | end |
| IQNINT | see NINT |

**Table 1-3  Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| ISHFT | Logical shift. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | `generic ISHFT(I,SHIFT)` |
| | `        INTEGER(1) function` |
| | `     BSHFT(I,SHIFT)` |
| | `           INTEGER(1) ::I,SHIFT` |
| | `      INTEGER(2) function` |
| | `     HSHFT(I,SHIFT)` |
| | `           INTEGER(2) ::I,SHIFT` |
| | `      INTEGER(2) function` |
| | `     IISHFT(I,SHIFT)` |
| | `           INTEGER(2) ::I,SHIFT` |
| | `      INTEGER(4) function` |
| | `     JISHFT(I,SHIFT)` |
| | `           INTEGER(4) ::I,SHIFT` |
| | `      INTEGER(8) function` |
| | `     KISHFT(I,SHIFT)` |
| | `           INTEGER(8) ::I,SHIFT` |
| | `     end` |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| ISHFTC | Circular shift of the rightmost bits. |
| | **Class.** generic elemental function |
| | **Summary.** |

```
generic ISHFTC(I,SHIFT,SIZE)
          INTEGER(1) function
        BSHFTC(I,SHIFT,SIZE)
              INTEGER(1) ::I,SHIFT
              INTEGER(1),OPTIONAL ::
        SIZE
          INTEGER(2) function
        HSHFTC(I,SHIFT,SIZE)
              INTEGER(2) ::I,SHIFT
              INTEGER(2),OPTIONAL ::
        SIZE
          INTEGER(2) function
        IISHFTC(I,SHIFT,SIZE)
              INTEGER(2) ::I,SHIFT
              INTEGER(2),OPTIONAL ::
        SIZE
          INTEGER(4) function
        JISHFTC(I,SHIFT,SIZE)
              INTEGER(4) ::I,SHIFT
              INTEGER(4),OPTIONAL ::
        SIZE
          INTEGER(8) function
        KISHFTC(I,SHIFT,SIZE)
              INTEGER(8) ::I,SHIFT
              INTEGER(8),OPTIONAL ::
        SIZE
        end
```

continued

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| ISIGN | Absolute value of A times the sign of B. |
|  | See also SIGN. |
|  | **Class.** generic nonstandard function |
|  | **Summary.** |
|  | ```
generic ISIGN(A,B)
              INTEGER(1) function BSIGN(A,B)
                  INTEGER(1) ::A,B
              INTEGER(2) function
          IISIGN(A,B)
                  INTEGER(2) ::A,B
              INTEGER(4) function
          JISIGN(A,B)
                  INTEGER(4) ::A,B
              INTEGER(8) function
          KISIGN(A,B)
                  INTEGER(8) ::A,B
              end
``` |
| ISNAN | Determine if a value is NaN. |
|  | **Class.** generic elemental nonstandard function |
|  | **Summary.** |
|  | ```
generic ISNAN(X)
              LOGICAL function ISNAN(X)
                  REAL(4) ::X
              LOGICAL function ISNAN(X)
                  REAL(8) ::X
              LOGICAL function ISNAN(X)
              end
``` |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| IXOR | Exclusive OR. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic IXOR(I,J)
                    INTEGER(1) function BIXOR(I,J)
                         INTEGER(1) ::I,J
                    INTEGER(2) function HIXOR(I,J)
                         INTEGER(2) ::I,J
                    INTEGER(2) function IIXOR(I,J)
                         INTEGER(2) ::I,J
                    INTEGER(4) function JIXOR(I,J)
                         INTEGER(4) ::I,J
                    INTEGER(8) function IXOR(I,J)
                         INTEGER(8) ::I,J
              end
``` |
| JIABS | see IABS |
| JIAND | see IAND |
| JIBCLR | see IBCLR |
| JIBITS | see IBITS |
| JIBSET | see IBSET |
| JIDIM | see IDIM |
| JIDINT | see IDINT |
| JIDNNT | see NINT |
| JIEOR | see IEOR |
| JIFIX | see INT |
| JINT | see INT |
| JIOR | see IOR |
| JIQINT | see IQINT |
| JIQNNT | see NINT |

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| JISHFT | see ISHFT |
| JISHFTC | see ISHFTC |
| JISIGN | see SIGN |
| JIXOR | see IXOR |
| JMAX0 | see "MAX(A1, A2, A3, ...)" |
| JMAX1 | see "MAX(A1, A2, A3, ...)" |
| JMIN0 | see "MIN(A1, A2, A3, ...)" |
| JMIN1 | see "MIN(A1, A2, A3, ...)" |
| JMOD | see "MOD(A, P)" |
| JNINT | see NINT |
| JNOT | see NOT |
| JNUM | Convert character to integer type. |
| | **Class.** specific elemental nonstandard function |
| | **Summary.** |
| | `INTEGER(4) function JNUM(I)` |
| | `                   CHARACTER ::I` |
| | `NOTE: The characters in the ASCII CHARACTER expression must be numeric, and the result must fit in an INTEGER(4) number.` |
| KIABS | see IABS |
| KIAND | see IAND |
| KIBCLR | see IBCLR |
| KIBITS | see IBITS |
| KIBSET | see IBSET |
| KIDIM | see IDIM |
| KIDINT | see IDINT |
| KIDNNT | see NINT |
| KIEOR | see IEOR |
| KIFIX | see INT |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| KIND | Return the kind type parameter of the argument. |
| | **Class.** generic inquiry function |
| | **Summary.** |

```
generic KIND(X)
                    INTEGER function KIND(X)
                         INTEGER(1) ::X
                    INTEGER function KIND(X)
                         INTEGER(2) ::X
                    INTEGER function KIND(X)
                         INTEGER(4) ::X
                    INTEGER function KIND(X)
                         INTEGER(8) ::X
                    INTEGER function KIND(X)
                       REAL(4) ::X
                    INTEGER function KIND(X)
                       REAL(8) ::X
                    INTEGER function KIND(X)
                       REAL(16)::X
                    INTEGER function KIND(X)
                       COMPLEX(4) ::X
                    INTEGER function KIND(X)
                       COMPLEX(8) ::X
                    INTEGER function KIND(X)
                       COMPLEX(16)::X
                    INTEGER function KIND(X)
                       LOGICAL(1) ::X
                    INTEGER function KIND(X)
                       LOGICAL(2) ::X
                    INTEGER function KIND(X)
                       LOGICAL(4) ::X
                    INTEGER function KIND(X)
                       LOGICAL(8) ::X
                  end
```

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| KINT | see INT |
| KIOR | see IOR |
| KIQINT | see IQINT |
| KIQNNT | see NINT |
| KISHFT | see ISHFT |
| KISHFTC | see ISHFTC |
| KISIGN | see SIGN |
| KMAX0 | see "MAX(A1, A2, A3, ...)" |
| KMAX1 | see "MAX(A1, A2, A3, ...)" |
| KMIN0 | see "MIN(A1, A2, A3, ...)" |
| KMIN1 | see "MIN(A1, A2, A3, ...)" |
| KMOD | see "MOD(A, P)" |
| KNINT | see NINT |
| KNOT | see NOT |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| LBOUND | Return lower bounds of an array. |

**Class.** generic inquiry function

**Summary.**

```
generic LBOUND(ARRAY,DIM)

!ARRAY must be array-valued, DIM must be
scalar

              INTEGER function
           LBOUND(ARRAY,DIM)
                 LOGICAL(1) ::ARRAY;
           INTEGER,OPTIONAL ::DIM
            INTEGER function
           LBOUND(ARRAY,DIM)
                 LOGICAL(2) ::ARRAY;
           INTEGER,OPTIONAL ::DIM
            INTEGER function
           LBOUND(ARRAY,DIM)
                 LOGICAL(4) ::ARRAY;
           INTEGER,OPTIONAL ::DIM
            INTEGER function
           LBOUND(ARRAY,DIM)
                 LOGICAL(8) ::ARRAY;
           INTEGER,OPTIONAL ::DIM
            INTEGER function
           LBOUND(ARRAY,DIM)
```

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| LBOUND (continued) | INTEGER(1) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function<br>LBOUND(ARRAY,DIM)<br>    INTEGER(2) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function<br>LBOUND(ARRAY,DIM)<br>    INTEGER(4) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function<br>LBOUND(ARRAY,DIM)<br> INTEGER(8) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function<br>LBOUND(ARRAY,DIM)<br>    REAL(4) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function<br>LBOUND(ARRAY,DIM)<br>    REAL(8) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>LBOUND(ARRAY,DIM)<br>    REAL(16) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function<br>LBOUND(ARRAY,DIM)<br>  COMPLEX(4) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| LBOUND (continued) | ```<br>        INTEGER function<br>LBOUND(ARRAY,DIM)<br>        COMPLEX(8) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>INTEGER function<br>LBOUND(ARRAY,DIM)<br>        COMPLEX(16) ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function LBOUND(ARRAY,<br>DIM)<br>        CHARACTER ::ARRAY;<br>INTEGER,OPTIONAL ::DIM<br>  INTEGER function LBOUND(ARRAY,<br>DIM)<br>        DERIVED_TYPE ::ARRAY;<br>INTEGER, OPTIONAL ::DIM<br>end<br>``` |
| LEN | Return the length of a character argument.<br>**Class.** generic inquiry function<br>**Summary.**<br>```<br>generic LEN(STRING)<br>        INTEGER function LEN(STRING)<br>                CHARACTER::STRING<br>        end<br>``` |
| LEN_TRIM | Length of a character string not including trailing blanks.<br>**Class.** generic inquiry function<br>**Summary.**<br>```<br>generic LEN_TRIM(STRING)<br>        INTEGER function<br>        LEN_TRIM(STRING)<br>                CHARACTER::STRING<br>        end<br>``` |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| LGE | Lexical greater than or equal to comparison for strings.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic LGE(STRING_A,STRING_B)<br>          LOGICAL function<br>          LGE(STRING_A,STRING_B)<br>               CHARACTER<br>          ::STRING_A,STRING_B<br>               end</pre> |
| LGT | Lexical greater than comparison for strings.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic LGT(STRING_A,STRING_B)<br>          LOGICAL function<br>          LGT(STRING_A,STRING_B)<br>               CHARACTER<br>          ::STRING_A,STRING_B<br>               end</pre> |
| LLE | Lexical less than or equal to comparison for strings.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic LLE(STRING_A,STRING_B)<br>          LOGICAL function<br>          LLE(STRING_A,STRING_B)<br>               CHARACTER<br>          ::STRING_A,STRING_B<br>               end</pre> |

**Table 1-3** **Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| LLT | Lexical less than comparison for strings.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic LLT(STRING_A,STRING_B)`<br>`        LOGICAL function`<br>`      LLT(STRING_A,STRING_B)`<br>`          CHARACTER`<br>`    ::STRING_A,STRING_B`<br>`    end` |
| LOC | Return the address of the argument.<br>**Class.** generic inquiry nonstandard function<br>**Summary.**<br>`INTEGER function LOC(X)`<br>`        ! X may be any one of the`<br>`        following:`<br>`          INTEGER(1) ::X`<br>`          INTEGER(2) ::X`<br>`          INTEGER(4) ::X`<br>`          INTEGER(8) ::X`<br>`          REAL(4) ::X`<br>`          REAL(8) ::X`<br>`          REAL(16)::X`<br>`          COMPLEX(4) ::X`<br>`          COMPLEX(8) ::X`<br>`          COMPLEX(16)::X`<br>`          LOGICAL(1) ::X`<br>`          LOGICAL(2) ::X`<br>`          LOGICAL(4) ::X`<br>`          LOGICAL(8) ::X` |

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| LOC (continued) | CHARACTER ::X |
| | *DERIVED TYPE* ::X |
| | **Note**: *For Itanium™-based applications, the returned type of the LOC function is INTEGER*8.* |
| LOG | Natural logarithm. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic LOG(X) |
| |       REAL function ALOG(X) |
| |         REAL ::X |
| |       DOUBLE PRECISION function DLOG(X) |
| |         DOUBLE PRECISION ::X |
| | REAL(16) function QLOG(X) |
| |         REAL(16) :: X |
| |       COMPLEX function CLOG(X) |
| |         COMPLEX ::X |
| |       DOUBLE COMPLEX function CDLOG(X) |
| |         DOUBLE COMPLEX ::X |
| |       DOUBLE COMPLEX function ZLOG(X) |
| |         DOUBLE COMPLEX ::X |
| |   COMPLEX(16) CQLOG(X) |
| |         COMPLEX(16) :: X |
| |       end |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| LOG10 | Common logarithm.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic LOG10(X)`<br>`        REAL function ALOG10(X)`<br>`            REAL ::X`<br>`        DOUBLE PRECISION function`<br>`    DLOG10(X)`<br>`            DOUBLE PRECISION ::X`<br>`      REAL(16) function QLOG10(X)`<br>`    REAL(16) :: X`<br>`    end` |

<div align="right">continued</div>

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| LOGICAL | Convert to logical type. |
| | **Class.** generic elemental function |
| | **Summary.** |

```
generic LOGICAL(L,KIND)
                LOGICAL(KIND) function
              LOGICAL(L,KIND)
                  LOGICAL(1) ::L
                  INTEGER,OPTIONAL ::KIND
                LOGICAL(KIND) function
              LOGICAL(L,KIND)
                  LOGICAL(2) ::L
                  INTEGER,OPTIONAL ::KIND
                LOGICAL(KIND) function
              LOGICAL(L,KIND)
                  LOGICAL(4) ::L
                  INTEGER,OPTIONAL ::KIND
                LOGICAL(KIND) function
              LOGICAL(L,KIND)
                  LOGICAL(8) ::L
                  INTEGER,OPTIONAL ::KIND
              LOGICAL function LOGICAL(L)
                  LOGICAL(1) ::L
              LOGICAL function LOGICAL(L)
                  LOGICAL(2) ::L
              LOGICAL function LOGICAL(L)
                  LOGICAL(4) ::L
              LOGICAL function LOGICAL(L)
                  LOGICAL(8) ::L
            end
```

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| LSHFT | Left shift. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic LSHFT(I,SHIFT)
              INTEGER(1) function
            LSHFT(I,SHIFT)
                 INTEGER(1) ::I,SHIFT
          INTEGER(2) function
            LSHFT(I,SHIFT)
                 INTEGER(2) ::I,SHIFT
          INTEGER(4) function
            LSHFT(I,SHIFT)
                 INTEGER(4) ::I,SHIFT
          INTEGER(8) function
            LSHFT(I,SHIFT)
                 INTEGER(8) ::I,SHIFT
            end
``` |
| | **Note**: SHIFT must be a positive integer expression of the same KIND as I. |

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| LSHIFT | Left shift. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic LSHIFT(I,SHIFT)
                  INTEGER(1) function
               LSHIFT(I,SHIFT)
                     INTEGER(1) ::I,SHIFT
                INTEGER(2) function
               LSHIFT(I,SHIFT)
                     INTEGER(2) ::I,SHIFT
                INTEGER(4) function
               LSHIFT(I,SHIFT)
                     INTEGER(4) ::I,SHIFT
                INTEGER(8) function
               LSHIFT(I,SHIFT)
                     INTEGER(8) ::I,SHIFT
                end
``` |
| | **Note**: SHIFT must be a positive integer expression of the same KIND as I. |
| MATMUL | Matrix multiply. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | ```
generic MATMAL(MATRIX_A,MATRIX_B)
``` |
| | **Notes**. |
| | MATRIX_A must be of numeric type or of logical type and must be array-valued and of rank one or two. |
| | MATRIX_B must be of numeric type if MATRIX_A is of numeric type, or of logical type if MATRIX_A is of logical type. If MATRIX_A has rank one, MATRIX_B must have rank two; if MATRIX_A has rank two, MATRIX_B must have rank one. |
| | If the arguments are logical, the result is of type logical. |
| | If the arguments are of numeric type, the result has numeric type. |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MAX | Maximum value. |

**Class.** generic elemental function

**Summary.**

```
generic MAX(A1,A2,...)
            REAL function
          AIMAX0(A1,A2,...)
                INTEGER(2) ::A1,A2,...
          REAL function
          AJMAX0(A1,A2,...)
                INTEGER(4) ::A1,A2,...
          REAL function
          AKMAX0(A1,A2,...)
                INTEGER(8) ::A1,A2,...
          REAL function AMAX0(A1,A2,...)
                INTEGER ::A1,A2,...
          INTEGER(2) function
        IMAX1(A1,A2,...)
                REAL ::A1,A2,...
          INTEGER(4) function
        JMAX1(A1,A2,...)
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MAX (continued) | REAL ::A1,A2,...<br>INTEGER function<br>MAX1(A1,A2,...)<br>REAL ::A1,A2,...<br>INTEGER(8) function<br>KMAX1(A1,A2,...)<br>REAL ::A1,A2,...<br>INTEGER(1) function<br>MAX(A1,A2,...)<br>INTEGER(1) ::A1,A2,...<br>INTEGER(2) function<br><br>IMAX0(A1,A2,...)<br>INTEGER(2) ::A1,A2,...<br>INTEGER(4) function<br>JMAX0(A1,A2,...)<br>INTEGER(4) ::A1,A2,...<br>INTEGER function<br>MAX0(A1,A2,...)<br>INTEGER ::A1,A2,<br><br>...<br>INTEGER(8) function<br>KMAX0(A1,A2,...)<br>INTEGER(8) ::A1,A2,...<br>REAL function AMAX1(A1,A2,...)<br>REAL ::A1,A2,...<br>DOUBLE PRECISION function<br>XMAX1(A1,A2,...)<br>DOUBLE   PRECISION<br>::A1,A2,...<br>end |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MAX0 | see "MAX(A1, A2, A3, ...)" |
| MAX1 | see "MAX(A1, A2, A3, ...)" |
| MAXEXPONENT | Return the maximum exponent in the model representing numbers of the same type and kind type parameter as the argument.<br><br>**Class.** generic inquiry function<br><br>**Summary.** |

```
generic MAXEXPONENT(X)
            INTEGER function
         MAXEXPONENT(X)
               REAL(4) ::X
          INTEGER function
         MAXEXPONENT(X)
               REAL(8) ::X
         INTEGER function MAXEXPONENT(X)
               REAL(16) :: X
         end
```

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MAXLOC | Return location of the first element of an array having the maximum value of elements identified by MASK. |

**Class.** generic transformational function

**Summary.**

```
generic MAXLOC(ARRAY,MASK)
                ! ARRAY must be array-valued
                  INTEGER function
                MAXLOC(ARRAY,MASK)
                      INTEGER(1) ::ARRAY;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER function
                MAXLOC(ARRAY,MASK)
                      INTEGER(2) ::ARRAY;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER function
                MAXLOC(ARRAY,MASK)
                      INTEGER(4) ::ARRAY;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER function
                MAXLOC(ARRAY,MASK)
                      INTEGER(8) ::ARRAY;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER function
                MAXLOC(ARRAY,MASK)
                      REAL(4) ::ARRAY;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER function
                MAXLOC(ARRAY,MASK)
                      REAL(8) ::ARRAY;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER function
```

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| `MAXLOC` (continued) | `        MAXLOC(ARRAY,MASK)`<br>`            REAL(16) ::ARRAY;`<br>`    LOGICAL,OPTIONAL ::MASK`<br>`      INTEGER function`<br>`    MAXLOC(ARRAY,MASK)`<br>`    end` |
| `MAXVAL` | Maximum value of elements of array along dimension `DIM` that correspond to `.TRUE.` elements of `MASK`.<br><br>**Class.** generic transformational function<br><br>**Summary.**<br><br>`generic MAXVAL(ARRAY,MASK,DIM)`<br>`          ! ARRAY must be array-valued`<br>`            INTEGER(1) function`<br>`    MAXVAL(ARRAY,DIM,MASK)`<br>`            INTEGER(1) ::ARRAY`<br>`            INTEGER,OPTIONAL ::DIM`<br>`    LOGICAL,OPTIONAL ::MASK`<br>`        INTEGER(2) function`<br>`    MAXVAL(ARRAY,DIM,MASK)`<br>`            INTEGER(2) ::ARRAY`<br>`            INTEGER,OPTIONAL ::DIM`<br>`    LOGICAL,OPTIONAL ::MASK`<br>`        INTEGER(4) function`<br>`    MAXVAL(ARRAY,DIM,MASK)`<br>`            INTEGER(4) ::ARRAY`<br>`            INTEGER,OPTIONAL ::DIM`<br>`    LOGICAL,OPTIONAL ::MASK`<br>`        INTEGER(8) function` |

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| MAXVAL (continued) | `MAXVAL(ARRAY,DIM,MASK)`<br>`        INTEGER(8) ::ARRAY`<br>`        INTEGER,OPTIONAL ::DIM`<br>`LOGICAL,OPTIONAL ::MASK`<br>`  REAL(4) function`<br>`MAXVAL(ARRAY,DIM,MASK)`<br>`        REAL(4) ::ARRAY`<br>`        INTEGER,OPTIONAL ::DIM`<br>`LOGICAL,OPTIONAL ::MASK`<br>`  REAL(8) function`<br>`MAXVAL(ARRAY,DIM,MASK)`<br>`        REAL(8) ::ARRAY`<br>`        INTEGER,OPTIONAL ::DIM`<br>`LOGICAL,OPTIONAL ::MASK`<br>`REAL(16) function`<br>`MAXVAL(ARRAY,DIM,MASK)`<br>`        REAL(16) ::ARRAY`<br>`        INTEGER,OPTIONAL ::DIM`<br>`LOGICAL,OPTIONAL ::MASK`<br>`end` |
| MCLOCK | Return time accounting for a program.<br>**Class.** specific inquiry nonstandard function<br>**Summary.**<br>`INTEGER function MCLOCK()` |
| MERGE | Choose alternative value according to the value of a mask.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic MERGE(TSOURCE,FSOURCE,MASK)`<br>`    TSOURCE` may be of any type.<br>`    FSOURCE` must be of the same type as `TSOURCE`.<br>`    MASK` must be type logical. |

continued

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| `MIN` | Minimum value. |

**Class.** generic elemental function

**Summary.**

```
generic MIN(A1,A2,...)
                REAL function
          AIMIN0(A1,A2,...)
                INTEGER(2) ::A1,A2,...
          REAL function
          AJMIN0(A1,A2,...)
                INTEGER(4) ::A1,A2,...
          REAL function
          AKMIN0(A1,A2,...)
                INTEGER(8) ::A1,A2,...
        REAL function AMIN0(A1,A2,...)
              INTEGER ::A1,A2,...
        INTEGER(2) function
        IMIN1(A1,A2,...)
              REAL ::A1,A2,...
        INTEGER(4) function
        JMIN1(A1,A2,...)
              REAL ::A1,A2,...
        INTEGER function
        MIN1(A1,A2,...)
              REAL ::A1,A2,...
        INTEGER(8) function
        KMIN1(A1,A2,...)
              REAL ::A1,A2,...
        INTEGER(1) function
        MIN(A1,A2,...)
              INTEGER(1) ::A1,A2,...
        INTEGER(2) function
        IMIN0(A1,A2,...)
```

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MIN (continued) | `INTEGER(2) ::A1,A2,...`<br>`INTEGER(4) function`<br>`JMIN0(A1,A2,...)`<br>`INTEGER(4) ::A1,A2,...`<br>`INTEGER function`<br>`MIN0(A1,A2,...)`<br>`INTEGER ::A1,A2,...`<br>`INTEGER(8) function`<br>`KMIN0(A1,A2,...)`<br>`INTEGER(8) ::A1,A2,...`<br>`REAL function AMIN1(A1,A2,...)`<br>`REAL ::A1,A2,...`<br>`DOUBLE PRECISION function`<br>`DMIN1(A1,A2,...)`<br>`DOUBLE   PRECISION`<br>`::A1,A2,...`<br>`REAL(16) function`<br>`QMIN1(A1,A2...)`<br>`REAL(16) :: A1,A2,...`<br>`end` |
| MIN0 | see "MIN(A1, A2, A3, ...)" |
| MIN1 | see "MIN(A1, A2, A3, ...)" |

<div align="right">continued</div>

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MINEXPONENT | Return the minimum exponent in the model representing numbers of the same type and kind type parameter as the argument.<br><br>**Class.** generic inquiry function<br><br>**Summary.**<br><br>```<br>generic MINEXPONENT(X)<br>          INTEGER function<br>      MINEXPONENT(X)<br>            REAL(4) ::X<br>        INTEGER function<br>      MINEXPONENT(X)<br>            REAL(8) ::XINTEGER<br>      function MINEXPONENT(X)<br>            REAL(16) :: X<br>        end<br>``` |

continued

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| MINLOC | Return location of the first element of an array having the minimum value of elements identified by MASK. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | ``generic MINLOC(ARRAY,MASK)`` |

```
                    ! ARRAY must be array-valued
                      INTEGER function
                    MINLOC(ARRAY,MASK)
                          INTEGER(1) ::ARRAY
                          LOGICAL,OPTIONAL ::MASK
                      INTEGER function
                    MINLOC(ARRAY,MASK)
                          INTEGER(2) ::ARRAY
                          LOGICAL,OPTIONAL ::MASK
                      INTEGER function
                    MINLOC(ARRAY,MASK)
                          INTEGER(4) ::ARRAY
                          LOGICAL,OPTIONAL ::MASK
                      INTEGER function
                    MINLOC(ARRAY,MASK)
                          INTEGER(8) ::ARRAY
                          LOGICAL,OPTIONAL ::MASK
                      INTEGER function
                    MINLOC(ARRAY,MASK)
                          REAL(4) ::ARRAY
                          LOGICAL,OPTIONAL ::MASK
                      INTEGER function
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| `MINLOC (continued)` | `MINLOC(ARRAY,MASK)`<br>　　`REAL(8) ::ARRAY`<br>　　`LOGICAL,OPTIONAL ::MASK`<br>`INTEGER function`<br>`MINLOC(ARRAY,MASK)`<br>　　`REAL(16) ::ARRAY`<br>`LOGICAL,OPTIONAL ::MASK` |
| `MINVAL` | Minimum value of elements of array along dimension `DIM` that correspond to `.TRUE.` elements of `MASK`.<br><br>**Class.** generic transformational function<br><br>**Summary.**<br><br>`generic MINVAL(ARRAY,MASK,DIM)`<br>　　`! ARRAY must be array-valued`<br>　`INTEGER(1) function`<br>`MINVAL(ARRAY,DIM,MASK)`<br>　　`INTEGER(1) ::ARRAY`<br>　　`INTEGER,OPTIONAL ::DIM;`<br>`LOGICAL,OPTIONAL ::MASK`<br>　`INTEGER(2) function`<br>`MINVAL(ARRAY,DIM,MASK)`<br>　　`INTEGER(2) ::ARRAY`<br>　　`INTEGER,OPTIONAL ::DIM;`<br>`LOGICAL,OPTIONAL ::MASK`<br>　`INTEGER(4) function`<br>`MINVAL(ARRAY,DIM,MASK)`<br>　　`INTEGER(4) ::ARRAY`<br>　　`INTEGER,OPTIONAL ::DIM;`<br>`LOGICAL,OPTIONAL ::MASK`<br>　`INTEGER(8) function`<br>`MINVAL(ARRAY,DIM,MASK)`<br>　　`INTEGER(8) ::ARRAY` |

continued

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MINVAL - (continued) | INTEGER,OPTIONAL ::DIM; LOGICAL,OPTIONAL ::MASK REAL(4) function MINVAL(ARRAY,DIM,MASK) REAL(4) ::ARRAY INTEGER,OPTIONAL ::DIM; LOGICAL,OPTIONAL ::MASK REAL(8) function MINVAL(ARRAY,DIM,MASK) REAL(8) ::ARRAY INTEGER,OPTIONAL ::DIM; LOGICAL,OPTIONAL ::MASK REAL(16) function MINVAL(ARRAY,DIM,MASK) REAL(16) ::ARRAY INTEGER,OPTIONAL ::DIM; LOGICAL,OPTIONAL ::MASK end |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MOD | Remainder function. |
| | **Class.** generic elemental function |
| | **Summary.** |

```
generic MOD(A,P)
                INTEGER(1) function BMOD(A,P)
                    INTEGER(1) ::A,P
                LOGICAL(1) function BMOD(A,P)
                     LOGICAL(1) ::A,P
                INTEGER(2) function HMOD(A,P)
                    INTEGER(2) ::A,P
                INTEGER(2) function IMOD(A,P)
                    INTEGER(2) ::A,P
                INTEGER(4) function JMOD(A,P)
                    INTEGER(4) ::A,P
                INTEGER(8) function KMOD(A,P)
                    INTEGER(8) ::A,P
              REAL function AMOD(A,P)
                  REAL ::A,P
              DOUBLE PRECISION function
            DMOD(A,P)
                  DOUBLE PRECISION ::A,P
            REAL(16) function QMOD(A,P)
                  REAL(16) :: A,P
            end
```

continued

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| MODULO | Modulo function. |
| | **Class.** generic elemental function |
| | **Summary.** |

```
generic MODULO(A,P)
              INTEGER(1) function
          MODULO(A,P)
                  INTEGER(1) ::A,P
          INTEGER(2) function
          MODULO(A,P)
                  INTEGER(2) ::A,P
          INTEGER(4) function
          MODULO(A,P)
                  INTEGER(4) ::A,P
          INTEGER(8) function
          MODULO(A,P)
                  INTEGER(8) ::A,P
        REAL(4) function MODULO(A,P)
              REAL(4) ::A,P
        REAL(8) function MODULO(A,P)
              REAL(8) ::A,P
          end
```

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| MVBITS | Copy a sequence of bits from one data object to another.<br>**Class.** generic elemental subroutine<br>**Summary.** |

```
generic MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)
              subroutine
        BMVBITS(FROM,FROMPOS,LEN,TO,TOPO
        S)
              INTEGER(1):: FROM,TO
              INTEGER :: FROMPOS, TOPOS,
        LEN
          subroutine
        HMVBITS(FROM,FROMPOS,LEN,TO,TOPO
        S)
              INTEGER(2):: FROM,TO
              INTEGER :: FROMPOS, TOPOS,
        LEN
          subroutine
        MVBITS(FROM,FROMPOS,LEN,TO,TOPOS
        )
              INTEGER(4):: FROM,TO
              INTEGER :: FROMPOS, TOPOS,
        LEN
          subroutine
        MVBITS(FROM,FROMPOS,LEN,TO,TOPOS
        )
              INTEGER(8):: FROM,TO
              INTEGER :: FROMPOS, TOPOS,
        LEN
        end
```

continued

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| NEAREST | Return the nearest different machine representable number in a given direction. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | <pre>generic NEAREST(X,S)<br>            REAL(4) function NEAREST(X,S)<br>                REAL(4) ::X,S<br>            REAL(8) function NEAREST(X,S)<br>                REAL(8) ::X,S<br>        REAL(16) function NEAREST(X,S)<br>                REAL(16) :: X,S<br>            end</pre> |
| NINT | Nearest integer. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | <pre>generic NINT(A,KIND)<br>            INTEGER(KIND) function<br>        NINT(A,KIND)<br>                REAL(4) ::A;<br>        INTEGER,OPTIONAL::KIND<br>            INTEGER(KIND) function<br>        NINT(A,KIND)<br>        REAL(8) ::A;</pre> |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| NINT (continued) | Nearest integer. |
| | **Class.** generic elemental function |
| | **Summary.** |

```
                    INTEGER,OPTIONAL::KIND
                      INTEGER function IDNINT(A)
                          DOUBLE PRECISION::A
                    INTEGER(2) function ININT(A)
                          REAL(4) A
                      INTEGER(4) function NINT(A)
                          REAL ::A
                      INTEGER(2) function IIDNNT(A)
                          DOUBLE PRECISION::A
                      INTEGER(2) function IIQNNT(A)
                           REAL(16) :: A
                      INTEGER(4) function JNINT(A)
                          REAL ::A
                      INTEGER(4) function JIDNNT(A)
                          DOUBLE PRECISION::A
                      INTEGER(4) function IQNINT(A)
                          REAL(16) :: A
                      INTEGER(8) function KNINT(A)
                          REAL ::A
                      INTEGER(8) function KIDNNT(A)
                          DOUBLE PRECISION::A
                      INTEGER(8) function KIQNNT(A)
                          REAL(16) A
                    end
```

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| NOT | Bitwise complement. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic NOT(I)
                    INTEGER(1) function BNOT(I)
                        INTEGER(1) ::I
                    INTEGER(2) function HNOT(I)
                        INTEGER(2) ::I
                    INTEGER(2) function INOT(I)
                        INTEGER(2) ::I
                    INTEGER(4) function JNOT(I)
                        INTEGER(4) ::I
                    INTEGER(8) function KNOT(I)
                        INTEGER(8) ::I
                end
``` |
| OR | Bitwise logical OR. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic OR(I,J)
                    INTEGER(1) function OR(I,J)
                        INTEGER(1) ::I,J
                    INTEGER(2) function OR(I,J)
                        INTEGER(2) ::I,J
                    INTEGER(4) function OR(I,J)
                        INTEGER(4) ::I,J
                    INTEGER(8) function OR(I,J)
                        INTEGER(8) ::I,J
                end
``` |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| PACK | Pack an array into an array of rank one under control of a mask. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | ```
generic PACK(ARRAY,MASK,VECTOR)
Notes.
``` |
| | ARRAY may be of any type; it must be array-valued. |
| | MASK must be of type logical and must be conformable with ARRAY. |
| | VECTOR is optional. It must be of same type as ARRAY and must be scalar. |
| | The result is an array of rank one with the same type as ARRAY. |
| PRECISION | Return the decimal precision in the model representing real numbers with the same kind type parameter as the argument. |
| | **Class.** generic inquiry function |
| | **Summary.** |
| | ```
generic PRECISION(X)
            INTEGER function PRECISION(X)
                REAL(4) ::X
            INTEGER function PRECISION(X)
                REAL(8) ::X
        INTEGER function PRECISION(X)
                REAL(16) :: X
            end
``` |

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| MM_PREFETCH | Prefetch the cache line containing R into the cache. Integer literal I selects the type of prefetch. |
| | **Class.** nonstandard function |
| | **Summary.** |
| | SUBROUTINE MM_PREFETCH(R, I)<br>    REAL*4 R<br>    INTEGER I<br><br>    I=0 ==> PREFETCHNTA<br>    I=1 ==> PREFETCHT0<br>    I=2 ==> PREFETCHT1<br>    I=3 ==> PREFETCHT2 |
| MM_PREFETCH | Prefetch the cache line containing D into the cache. Integer literal I selects the type of prefetch. |
| | **Class.** nonstandard function |
| | **Summary.** |
| | SUBROUTINE MM_PREFETCH(D, I)<br>    DOUBLE PRECISION D<br>    INTEGER I<br><br>    I=0 ==> PREFETCHNTA<br>    I=1 ==> PREFETCHT0<br>    I=2 ==> PREFETCHT1<br>    I=3 ==> PREFETCHT2 |

continued

**Table 1-3    Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| PRESENT | Determine whether an optional argument is present. |
| | **Class.** generic inquiry function |
| | **Summary.** |
| | `generic PRESENT(A)` |
| | Notes. |
| |        A may be of any type. It must be an optional argument of the procedure in which PRESENT function reference appears. |
| |        The result has type LOGICAL. |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| PRODUCT | Product of all elements of an array along dimension DIM corresponding to the .TRUE. elements of MASK. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | generic PRODUCT(ARRAY,DIM,MASK) |

```
                !ARRAY must be array-valued, DIM
                must be scalar
                  INTEGER(1) function
                PRODUCT(ARRAY,DIM,MASK)
                      INTEGER(1) ::ARRAY
                      INTEGER,OPTIONAL ::DIM;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER(2) function
                PRODUCT(ARRAY,DIM,MASK)
                      INTEGER(2) ::ARRAY
                      INTEGER,OPTIONAL ::DIM;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER(4) function
                PRODUCT(ARRAY,DIM,MASK)
                      INTEGER(4) ::ARRAY
                      INTEGER,OPTIONAL ::DIM;
                LOGICAL,OPTIONAL ::MASK
                  INTEGER(8) function
                PRODUCT(ARRAY,DIM,MASK)
                      INTEGER(8) ::ARRAY
                      INTEGER,OPTIONAL ::DIM;
                LOGICAL,OPTIONAL ::MASK
                  REAL(4) function
                PRODUCT(ARRAY,DIM,MASK)
                      REAL(4) ::ARRAY
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| `PRODUCT` (continued) | `INTEGER,OPTIONAL ::DIM;`<br>`LOGICAL,OPTIONAL ::MASK`<br>`REAL(8) function`<br>`PRODUCT(ARRAY,DIM,MASK)`<br>`REAL(8) ::ARRAY`<br>`INTEGER,OPTIONAL ::DIM;`<br>`LOGICAL,OPTIONAL ::MASK`<br>`LOGICAL,OPTIONAL ::MASK`<br>`end` |
| `QABS` | see "ABS(A)" |
| `QACOS` | see "ACOS(X)" |
| `QACOSD` | see "ACOSD(X)" |
| `QACOSH` | see "ACOSH(X)" |
| `QASIN` | see "ASIN(X)" |
| `QASIND` | see "ASIND(X)" |
| `QASINH` | see "ASINH(X)" |
| `QATAN` | see "ATAN(X)" |
| `QATAN2` | see "ATAN2(Y, X)" |
| `QATAN2D` | see "ATAN2D(Y, X)" |
| `QATAND` | see "ATAND(X)" |
| `QATANH` | see "ATANH(X)" |
| `QCOS` | see "COS(X)" |
| `QCOSD` | see "COSD(X)" |
| `QCOSH` | see "COSH(X)" |
| `QDIM` | see "DIM(X, Y)" |
| `QEXP` | see "EXP(X)" |
| `QINT` | see "AINT(A, KIND)" |
| `QLOG` | see "LOG(X)" |
| `QLOG10` | see "LOG10(X)" |

**Table 1-3       Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| QMAX1 | see "MAX(A1, A2, A3, ...)" |
| QMIN1 | see "MIN(A1, A2, A3, ...)" |
| QMOD | see "MOD(A, P)" |
| QNINT | see "ANINT(A, KIND)" |
| QPROD | Double precision product. |
|  | **Class.** generic elemental nonstandard function |
|  | **Summary.** |
|  | generic QPROD(X,Y) |
|  |                    DOUBLE PRECISION ::X,Y |
|  |            end |
| QSIGN | see "SIGN(A, B)" |
| QSIN | see "SIN(X)" |
| QSIND | see "SIND(X)" |
| QSINH | see "SINH(X)" |
| QSQRT | see "SQRT(X)" |
| QTAN | see "TAN(X)" |
| QTAND | see "TAND(X)" |
| QTANH | see "TANH(X)" |

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| RADIX | Return the base of the model representing numbers of the same type and kind type parameter as the argument. |
| | **Class.** generic inquiry function |
| | **Summary.** |
| | ``` generic RADIX(X) |
| | INTEGER function RADIX(X) |
| | INTEGER(1) ::X |
| | INTEGER function RADIX(X) |
| | INTEGER(2) ::X |
| | INTEGER function RADIX(X) |
| | INTEGER(4) ::X |
| | INTEGER function RADIX(X) |
| | INTEGER(8) ::X |
| | INTEGER function RADIX(X) |
| | REAL(4) ::X |
| | INTEGER function RADIX(X) |
| | REAL(8) ::X |
| | end ``` |
| RANDOM_NUMBER | Generate pseudorandom number in the range of 0. to 1. |
| | **Class.** generic subroutine |
| | **Summary.** |
| | ``` generic RANDOM_NUMBER(HARVEST) |
| | subroutine |
| | RANDOM_NUMBER(HARVEST) |
| | REAL(4) ::HARVEST |
| | subroutine |
| | RANDOM_NUMBER(HARVEST) |
| | REAL(8) ::HARVEST |
| | end ``` |

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| RANDOM_SEED | Restart or query the pseudorandom number generator used by RANDOM_NUMBER. |
|  | **Class.** generic subroutine |
|  | **Summary.** |
|  | ```generic RANDOM_SEED(SIZE,PUT,GET)``` |
|  | ```! There must be exactly one or no arguments``` |
|  | ```!``` |
|  | ```! PUT and GET must be a rank-one and``` |
|  | ```! must be of sufficient size   subroutine RANDOM_SEED(SIZE,PUT,GET)``` |
|  | ```        INTEGER,OPTIONAL ::SIZE,PUT,GET``` |
|  | ```end``` |

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| RANGE | Return the decimal exponent range in the model representing integer or real numbers with the same kind type parameter as the argument. |

**Class.** generic inquiry function

**Summary.**

```
generic RANGE(X)
              INTEGER function RANGE(X)
                  INTEGER(1) ::X
              INTEGER function RANGE(X)
                  INTEGER(2) ::X
              INTEGER function RANGE(X)
                  INTEGER(4) ::X
              INTEGER function RANGE(X)
                  INTEGER(8) ::X
              INTEGER function RANGE(X)
                 REAL(4) ::X
              INTEGER function RANGE(X)
                 REAL(8) ::X
           end
```

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| REAL | Convert to real type. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic REAL(A,KIND) |

```
                        REAL function FLOAT(A)
                            INTEGER ::A
                        REAL function REAL(A)
                            INTEGER(1) ::A
                        REAL function FLOATI(A)
                            INTEGER(2) ::A
                        REAL function FLOATJ(A)
                            INTEGER(4) ::A
                        REAL function FLOATK(A)
                            INTEGER(8) ::A
                        REAL function SNGL(A)
                            REAL ::A
                        REAL function REAL(A)
                            REAL(4) ::A
                        REAL function REAL(A)
                            REAL(8) ::A
                        REAL function REAL(A)
                            COMPLEX(4) ::A
                        REAL function REAL(A)
                            COMPLEX(8) ::A
                    end
```

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| REPEAT | Concatenate several copies of a string.<br>**Class.** generic transformational function<br>**Summary.**<br>`generic REPEAT(STRING,NCOPIES)`<br>`            CHARACTER function`<br>`           REPEAT(STRING,NCOPIES)`<br>`                CHARACTER ::STRING`<br>`                INTEGER ::NCOPIES`<br>`           end` |
| RESHAPE | Construct an array of a specified shape from the elements of a given array.<br>**Class.** generic transformational function<br>**Summary.**<br>`generic RESHAPE(SOURCE,SHAPE,PAD,ORDER)`<br>`Notes.`<br>`          SOURCE may be of any type. It`<br>`          must be array-valued.`<br>`          SHAPE must be of type integer,`<br>`          rank one and constant size.`<br>`          PAD is optional. It must have the`<br>`          same type as SOURCE.`<br>`          ORDER is optional. It must be of`<br>`          type integer.`<br>`          The result has the same type as`<br>`          SOURCE.` |
| RNUM | Convert character to real type.<br>**Class.** specific elemental nonstandard function<br>**Summary.**<br>`          REAL function RNUM(I)`<br>`              CHARACTER ::I` |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| RRSPACING | Return the reciprocal of the relative spacing of model numbers near the argument value. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic RRSPACING(X)
            REAL(4) function RRSPACING(X)
                REAL(4) ::X
            REAL(8) function RRSPACING(X)
                REAL(8) ::X
        REAL(16) RRSPACING(X)
                REAL(16) :: X
            end
``` |
| RSHFT | Bitwise right shift. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic RSHIFT(I,SHIFT)
            INTEGER(1) function
        RSHFT(I,SHIFT)
                INTEGER(1) ::I,SHIFT
            INTEGER(2) function
        RSHFT(I,SHIFT)
                INTEGER(2) ::I,SHIFT
            INTEGER(4) function
        RSHFT(I,SHIFT)
                INTEGER(4) ::I,SHIFT
            INTEGER(8) function
        RSHFT(I,SHIFT)
                INTEGER(8) ::I,SHIFT
            end
``` |
| | **Note:** SHIFT must be a positive integer of the same KIND as I. |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| RSHIFT | Bitwise right shift. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic RSHIFT(I,SHIFT)
                INTEGER(1) function
            RSHIFT(I,SHIFT)
                    INTEGER(1) ::I,SHIFT
            INTEGER(2) function
            RSHIFT(I,SHIFT)
                    INTEGER(2) ::I,SHIFT
            INTEGER(4) function
            RSHIFT(I,SHIFT)
                    INTEGER(4) ::I,SHIFT
            INTEGER(8) function
            RSHIFT(I,SHIFT)
                    INTEGER(8) ::I,SHIFT
            end
``` |
| | **Note:** SHIFT must be a positive integer of the same KIND as I. |
| SCALE | Return X*(b**I) where b is the base in the model representation of X. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic SCALE(X,I)
            REAL(4) function SCALE(X,I)
                REAL(4) ::X; INTEGER ::I
            REAL(8) function SCALE(X,I)
                REAL(8) ::X; INTEGER ::I
        REAL(16) function SCALE(X,I)
                REAL(16) :: X
                INTEGER::I
            end
``` |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| SCAN | Scan a string for any one of the characters in a set of characters.<br>**Class.** generic elemental function<br>**Summary.**<br><pre>generic SCAN(STRING,SET,BACK)<br>            INTEGER function<br>          SCAN(STRING,SET,BACK)<br>              CHARACTER ::STRING,SET<br>              LOGICAL,OPTIONAL :: BACK<br>          end</pre> |
| SELECTED_INT_KIND | Return the value of the kind type parameter of an integer data type that represents all integer values n with<br>$-10**r < n < 10**r$.<br>**Class.** generic transformational function<br>**Summary.**<br><pre>generic SELECTED_INT_KIND(R)<br>            INTEGER function<br>          SELECTED_INT_KIND(R)<br>                INTEGER ::R<br>          end</pre> |
| SELECTED_REAL_KIND | Return the value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range of at least R.<br>**Class.** generic transformational function<br>**Summary.**<br><pre>generic SELECTED_REAL_KIND(P,R)<br>          ! At least one argument must be<br>          present.<br>            INTEGER function<br>          SELECTED_REAL_KIND(P,R)<br>              INTEGER,OPTIONAL ::P,R<br>          end</pre> |

continued

**Table 1-3     Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| SET_EXPONENT | Return the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic SET_EXPONENT(X,I)
          REAL(4) function
        SET_EXPONENT(X,I)
              REAL(4) ::X; INTEGER ::I
          REAL(8) function
        SET_EXPONENT(X,I)
              REAL(8) ::X; INTEGER ::I
        REAL(16) function
        SET_EXPONENT(X,I)
              REAL(16) :: X
              INTEGER :: I
        end
``` |
| SHAPE | Return the shape of an array or a scalar. |
| | **Class.** generic inquiry function |
| | **Summary.** |
| | ```
generic SHAPE(SOURCE)
Notes.
        SOURCE may be of any type.
        The result is of type integer and
        is an array of rank one.
``` |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| SIGN | Absolute value of A times the sign of B. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | generic SIGN(A,B) |
| | INTEGER(1) function BSIGN(A,B) |
| | INTEGER(1) ::A,B |
| | INTEGER(2) function HSIGN(A,B) |
| | INTEGER(2) ::A,B |
| | INTEGER(2) function IISIGN(A,B) |
| | INTEGER(2) ::A,B |
| | INTEGER(4) function ISIGN(A,B) |
| | INTEGER(4) ::A,B |
| | INTEGER(4) function JSIGN(A,B) |
| | INTEGER(4) ::A,B |
| | INTEGER(8) function KISIGN(A,B) |
| | INTEGER(8) ::A,B |
| | REAL function SIGN(A,B) |
| | REAL ::A,B |
| | DOUBLE PRECISION function DSIGN(A,B) |
| | DOUBLE PRECISION ::A,B |
| | REAL(16) function QSIGN(A,B) |
| | REAL(16) :: A,B |
| | end |

continued

**Table 1-3**    **Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| SIN | Sine function that accepts input in radians. |
| | **Class.** generic elemental function |
| | **Summary.** |

```
generic SIN(X)
              REAL function SIN(X)
                  REAL ::X
              DOUBLE PRECISION function
          DSIN(X)
                  DOUBLE PRECISION ::X
              COMPLEX function CSIN(X)
                  COMPLEX ::X
            DOUBLE COMPLEX function
          CDSIN(X)
                  DOUBLE COMPLEX ::X
            DOUBLE COMPLEX function
          ZSIN(X)
                  DOUBLE COMPLEX ::X
               REAL(16) function QSIN(X)
                  REAL(16) X
            COMPLEX(16) function CQSIN(X)
                  COMPLEX(16) ::X
          end
```

continued

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| SIND | Sine function that accepts input in degrees.<br>**Class.** generic elemental nonstandard function<br>**Summary.**<br>generic SIND(X)<br>            REAL function SIND(X)<br>                REAL ::X<br>                DOUBLE PRECISION function DSIND(X)<br>                    DOUBLE PRECISION ::X<br>                end |
| SINH | Hyperbolic sine function.<br>**Class.** generic elemental function<br>**Summary.**<br>generic SINH(X)<br>            REAL function SINH(X)<br>                REAL ::X<br>            DOUBLE PRECISION function DSINH(X)<br>                DOUBLE PRECISION ::X<br>            REAL(16) function QSINH(X)<br>                REAL(16) :: X<br>            end |

continued

**Table 1-3** **Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| SIZE | Return the number of elements of an array or the extent along a specified dimension.<br><br>**Class.** generic non-standard inquiry function<br><br>**Summary.**<br><br>`generic SIZE(ARRAY,DIM)`<br><br>`Notes.`<br><br>`ARRAY may be of any type. It must be array-valued.`<br><br>`DIM is optional. It must be of type integer and must not be array-valued.`<br><br>`The result is of type integer.` |
| SIZEOF | Return the number of bytes of storage used by the argument.<br><br>**Class.** generic inquiry function<br><br>**Summary.**<br><br>`generic SIZEOF(A)`<br><br>**Note**. A may be of any type. The result is of type integer. If A is a pointer, the function returns the size of what A points to rather than the size of A itself. |
| SNGL | see REAL |
| SNGLQ | see REAL |
| SPACING | Return the absolute spacing of model numbers near the argument value.<br><br>**Class.** generic elemental function<br><br>**Summary.**<br><br>`generic SPACING(X)`<br><br>`REAL(4) function SPACING(X)`<br>`REAL(4) ::X`<br>`REAL(8) function SPACING(X)`<br>`REAL(8) ::X`<br>`end` |

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| SPREAD | Replicate an array by adding a dimension. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | generic SPREAD(SOURCE,DIM,NCOPIES) |
| | Notes. |
| | SOURCE may be of any type. |
| | DIM must be of type integer and must be scalar. |
| | NCOPIES must be of type integer and must be scalar. |
| | The result has the same type as SOURCE. |

continued

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| SQRT | Square root. |

**Class.** generic elemental function

**Summary.**

```
generic SQRT(X)
            REAL function SQRT(X)
                REAL ::X
            DOUBLE PRECISION function
        DSQRT(X)
                DOUBLE PRECISION ::X
            COMPLEX function CSQRT(X)
                COMPLEX ::X
            DOUBLE COMPLEX function
        CDSQRT(X)
                DOUBLE COMPLEX ::X
            DOUBLE COMPLEX function
        ZSQRT(X)
                DOUBLE COMPLEX ::X
        REAL(16) function QSQRT(X)
                REAL(16) :: X
        COMPLEX(16) function CQSQRT(X)
                COMPLEX(16) :: X
            end
```

continued

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| SUM | Sum all the elements of ARRAY along dimension DIM corresponding to all the .TRUE. elements of MASK. |
| | **Class.** generic transformational function |
| | **Summary.** |
| | ```
generic SUM(ARRAY,MASK,DIM)
            ! ARRAY must be array-valued
              INTEGER(1) function
            SUM(ARRAY,DIM,MASK)
                  INTEGER(1) ::ARRAY
                  INTEGER,OPTIONAL ::DIM;
            LOGICAL,OPTIONAL ::MASK
              INTEGER(2) function
            SUM(ARRAY,DIM,MASK)
                  INTEGER(2) ::ARRAY
                  INTEGER,OPTIONAL ::DIM;
            LOGICAL,OPTIONAL ::MASK
              INTEGER(4) function
            SUM(ARRAY,DIM,MASK)
                  INTEGER(4) ::ARRAY
                  INTEGER,OPTIONAL ::DIM;
            LOGICAL,OPTIONAL ::MASK
              INTEGER(8) function
            SUM(ARRAY,DIM,MASK)
                  INTEGER(8) ::ARRAY
                  INTEGER,OPTIONAL ::DIM;
            LOGICAL,OPTIONAL ::MASK
              REAL(4) function
            SUM(ARRAY,DIM,MASK)
                  REAL(4) ::ARRAY;
                  INTEGER,OPTIONAL ::DIM;
            LOGICAL,OPTIONAL ::MASK
            end
``` |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| SUM (continued) | REAL(8) function SUM(ARRAY,DIM,MASK)<br><br>REAL(8) ::ARRAY<br><br>INTEGER,OPTIONAL ::DIM; LOGICAL,OPTIONAL ::MASK<br><br>REAL(16) function SUM(ARRAY,DIM,MASK)<br><br>REAL(16) ::ARRAY<br><br>INTEGER,OPTIONAL ::DIM; LOGICAL,OPTIONAL ::MASK |
| SYSTEM_CLOCK | Return integer data from a real-time clock.<br><br>**Class.** generic subroutine<br><br>**Summary.**<br><br>generic SYSTEM_CLOCK(COUNT,COUNT_RATE,COUNT_MAX)<br><br>subroutine SYSTEM_CLOCK(COUNT,COUNT_RATE,COUNT_MAX)<br><br>INTEGER,OPTIONAL ::COUNT,COUNT_RATE,COUNT_MAX<br><br>end |

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| TAN | Tangent in radians. |
| | **Class.** generic elemental function |
| | **Summary.** |
| | ```
generic TAN(X)
         REAL function TAN(X)
             REAL ::X
         DOUBLE PRECISION function
DTAN(X)
             DOUBLE PRECISION ::X
         COMPLEX function CTAN(X)
             COMPLEX ::X
     DOUBLE COMPLEX function
ZTAN(X)
             DOUBLE COMPLEX ::X
      REAL(16) function QTAN(X)
             REAL(16) :: X
       COMPLEX(16) function CQTAN(X)
             COMPLEX(16) :: X
     end
``` |
| TAND | Tangent function that accepts input in degrees. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | ```
generic TAND(X)
         REAL function TAND(X)
             REAL ::X
         DOUBLE PRECISION function
DTAND(X)
             DOUBLE PRECISION ::X
     REAL(16) function QTAND(X)
             REAL(16) :: X
     end
``` |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| TANH | Hyperbolic tangent.<br>**Class.** generic elemental function<br>**Summary.**<br><br>```<br>generic TANH(X)<br>            REAL function TANH(X)<br>                REAL ::X<br>              DOUBLE PRECISION function<br>          DTANH(X)<br>                  DOUBLE PRECISION ::X<br>          REAL(16) function QTANH<br>                  REAL(16) QTANH<br>            end<br>``` |
| TINY | Return the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.<br>**Class.** generic inquiry function<br>**Summary.**<br><br>```<br>generic TINY(X)<br>            REAL(4) function TINY(X)<br>                REAL(4) ::X<br>              REAL(8) function TINY(X)<br>                  REAL(8) ::X<br>           REAL(16) function TINY(X)<br>                  REAL(16) :: X<br>            end<br>``` |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| TRANSFER | Return a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.<br><br>**Class.** generic transformational function<br>**Summary.**<br>generic TRANSFER(SOURCE,MOLD,SIZE)<br>Notes.<br>SOURCE may be of any type.<br>MOLD may be of any type.<br>SIZE is optional. It must be a scalar of type integer.<br>The result has the same type as MOLD. |
| TRANSPOSE | Transpose an array of rank two.<br>**Class.** generic transformational function<br>**Summary.**<br>generic TRANSPOSE(MATRIX)<br>Notes.<br>MATRIX may be of any type. It must have rank two.<br>The result has the same type as MATRIX. |
| TRIM | Return the argument with trailing blank characters removed.<br>**Class.** generic transformational function<br>**Summary.**<br>generic TRIM(STRING)<br>CHARACTER function TRIM(STRING)<br>CHARACTER ::STRING<br>end |

**Table 1-3     Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| UBOUND | Returns upper bounds of an array.<br>**Class.** generic inquiry function<br>**Summary.**<br>`generic UBOUND(ARRAY,DIM)`<br>    `!ARRAY must be array-valued, DIM must be scalar`<br>    `INTEGER function UBOUND(ARRAY,DIM)`<br>      `LOGICAL(1) ::ARRAY; INTEGER,OPTIONAL ::DIM`<br>    `INTEGER function UBOUND(ARRAY,DIM)`<br>      `LOGICAL(2) ::ARRAY; INTEGER,OPTIONAL ::DIM`<br>    `INTEGER function UBOUND(ARRAY,DIM)`<br>      `LOGICAL(4) ::ARRAY; INTEGER,OPTIONAL ::DIM`<br>    `INTEGER function UBOUND(ARRAY,DIM)`<br>      `LOGICAL(8) ::ARRAY; INTEGER,OPTIONAL ::DIM`<br>    `INTEGER function UBOUND(ARRAY,DIM)`<br>      `INTEGER(1) ::ARRAY; INTEGER,OPTIONAL ::DIM`<br>    `INTEGER function UBOUND(ARRAY,DIM)`<br>      `INTEGER(2) ::ARRAY; INTEGER,OPTIONAL ::DIM`<br>`INTEGER function UBOUND(ARRAY,DIM)`<br>      `INTEGER(4) ::ARRAY; INTEGER,OPTIONAL ::DIM` |

continued

**Table 1-3      Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
|---|---|
| UBOUND (continued) | ``` INTEGER function UBOUND(ARRAY,DIM)     INTEGER(8) ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |
| | ``` INTEGER function UBOUND(ARRAY,DIM)     REAL(4) ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |
| | ``` INTEGER function UBOUND(ARRAY,DIM)     REAL(8) ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |
| | ``` INTEGER function UBOUND(ARRAY,DIM)     REAL(16) ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |
| | ``` INTEGER function UBOUND(ARRAY,DIM)     COMPLEX(4) ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |
| | ``` INTEGER function UBOUND(ARRAY,DIM)     COMPLEX(8) ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |
| | ``` INTEGER function UBOUND(ARRAY,DIM)     COMPLEX(16) ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |
| | ``` INTEGER function UBOUND(ARRAY,DIM)     CHARACTER ::ARRAY; INTEGER,OPTIONAL ::DIM ``` |

**Table 1-3      Generic and Specific Intrinsic Procedures**  (continued)

| Intrinsic Procedure | Description |
|---|---|
| UBOUND (continued) | ``` INTEGER function UBOUND(ARRAY,DIM) DERIVED_TYPE ::ARRAY; INTEGER,OPTIONAL ::DIM end ``` |
| UNPACK | Unpack an array of rank one into an array under control of a mask.<br>**Class.** generic transformational function<br>**Summary.**<br>`generic UNPACK(VECTOR,MASK,FIELD)`<br>Notes.<br><br>VECTOR may be of any type. It must have rank one.<br>MASK must have logical type. It must be array-valued.<br>FIELD must be the same type as VECTOR.<br>The result has the same type as VECTOR. |
| VERIFY | Verify that a set of a characters contains all the characters in a string by identifying the position of the first character that does not appear in a given set of characters.<br>**Class.** generic elemental function<br>**Summary.**<br>`generic VERIFY(STRING,SET,BACK)`<br>``` INTEGER function VERIFY(STRING,SET,BACK) CHARACTER ::STRING, SET LOGICAL,OPTIONAL ::BACK end ``` |

**Table 1-3    Generic and Specific Intrinsic Procedures** (continued)

| Intrinsic Procedure | Description |
| --- | --- |
| XOR | Bitwise exclusive OR. |
| | **Class.** generic elemental nonstandard function |
| | **Summary.** |
| | generic XOR(I,J) |
| |         INTEGER(1) function XOR(I,J) |
| |            INTEGER(1) ::I,J |
| |         INTEGER(2) function XOR(I,J) |
| |            INTEGER(2) ::I,J |
| |         INTEGER(4) function XOR(I,J) |
| |            INTEGER(4) ::I,J |
| |         INTEGER(8) function XOR(I,J) |
| |            INTEGER(8) ::I,J |
| | end |
| | **Note:** this function can also be specified in all its forms as IXOR |
| ZABS | see ABS |
| ZCOS | see COS |
| ZEXP | see EXP |
| ZLOG | see "LOG(X)" |
| ZSIN | see SIN |
| ZSQRT | see SQRT |
| ZTAN | see TAN |

## Intrinsic Procedure Specifications

This section contains detailed specifications of Intel Fortran intrinsic procedures, which are listed in alphabetical order.

For a summary of all generic as well as specific intrinsic procedures, see Table 1-3 in the section "Generic and Specific Intrinsic Summary".

All of the intrinsic procedures in this section are generic. This means that each intrinsic procedure may be called with more than one argument type/kind/rank pattern.

In many cases, the kind and type of intrinsic function results are the same as that of the "principal" argument. For example, the `SIN` function may be called with any kind of real argument or any kind of complex argument, and the result has the type and kind of the argument.

Intrinsic procedure references may use keywords, in which case the actual argument expression is preceded by the dummy argument name (the argument keyword) and an "=" symbol. These argument keywords are shown in the following descriptions of the procedures.

Some intrinsic procedure's arguments are optional. Optional arguments are noted as such in the following descriptions.

## ABS(A)

### Description

Absolute value.

### Class

Elemental function.

### Argument

A must be of type integer, real, or complex.

### Result Type and Type Parameter

The same as A except that if A is complex, the result is real.

### Result Value

If A is of type integer or real, the value of the result is |A|.

If A is complex with value $(x, y)$, the result is equal to a processor-dependent approximation to $\sqrt{x^2 + y^2}$

### Examples

```
ABS(-1) has the value 1.
ABS(-1.5) has the value 1.5.
ABS((3.0, 4.0)) has the value 5.0.
```

# ACHAR(I)

*Converts an integer to an ASCII representation*

### Description

Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the IACHAR function.

### Class

Elemental function.

### Argument

I must be of type integer.

### Result Type and Type Parameter

Character of length one with kind type parameter value KIND('A').

**Result Value**

If `I` has a value in the range $0 \le I \le 127$, the result is the character in position `I` of the ASCII collating sequence, provided the processor is capable of representing that character; otherwise, the result is processor-dependent.

If the processor is not capable of representing both uppercase and lowercase letters and `I` corresponds to a letter in a case that the processor is not capable of representing, the result is the letter in the case that the processor is capable of representing.

`ACHAR(IACHAR(C))` must have the value `C` for any character `C` capable of representation in the processor.

**Examples**

```
ACHAR(88) is 'X'.
ACHAR(42) is '*'.
```

# ACOS(X)

**Description**

Arccosine (inverse cosine) function in radians.

**Class**

Elemental function.

**Argument**

`X` must be of type real with a value that satisfies the inequality $|X| \le 1$.

**Result Type and Type Parameter**

Same as `X`.

**Result Value**

```
The result has a value equal to a processor-dependent
approximation to arccos(X), expressed in radians.  It
lies in the range 0 ≤ ACOS(X) ≤ π.
```

**Examples**

```
ACOS(0.54030231) has the value 1.0.
ACOS(.1_HIGH) has the value 1.4706289056333 with
kind HIGH.
```

# ACOSD(X)

### Description

Arccosine (inverse cosine) in degrees.

### Class

Elemental nonstandard function.

### Argument

X must be of type real with a value that satisfies the inequality $|X| \leq 1$.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to arccos(X), expressed in degrees. It lies in the range $0 \leq \text{ACOSD(X)} \leq 180$.

### Examples

```
ACOSD(0.0000001) has the value 89.99999.
ACOSD(0.5) has the value 60.0.
ACOSD(-1.0) has the value 180.0.
```

# ACOSH(X)

### Description

Hyperbolic arccosine of radians.

### Class

Elemental nonstandard function.

### Argument

X must be of type real with a value $X \geq 1$.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to the hyperbolic arccosine of X. It lies in the range $0 \leq \text{ACOSH(X)}$.

### Examples

```
ACOSH(1.0) has the value 0.0.
ACOSH(180.0) has the value 5.8861.
ACOSH(0.0) has the value NaN (not a number).
```

## ADJUSTL(STRING)

### Description

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

### Class

Elemental function.

### Argument

STRING must be of type character.

### Result Type

Character of the same length and kind type parameter as STRING.

### Result Value

The value of the result is the same as STRING except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

### Example

ADJUSTL('ƀƀWORD') is 'WORDƀƀ'.

# ADJUSTR(STRING)

### Description

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

### Class

Elemental function.

### Argument

STRING must be of type character.

### Result Type

Character of the same length and kind type parameter as STRING.

### Result Value

The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

### Examples

ADJUSTR('WORDbb') has the value 'bbWORD'.

# AIMAG(Z)

### Description

Imaginary part of a complex number.

### Class

Elemental function.

### Argument

Z must be of type complex.

### Result Type and Type Parameter

Real with the same kind type parameter as Z.

### Result Value

If Z has the value $(x, y)$, the result has value $y$.

### Examples

AIMAG((2.0, 3.0)) has the value 3.0.

AIMAG((2.0_HIGH, 3.0)) has the value 3.0 with kind HIGH; the parts of a complex literal constant have the same precision, which is that of the part with the greatest precision.

# AINT(A, KIND)

### Optional Argument

KIND

### Description

Truncation to a whole number.

### Class

Elemental function.

### Arguments

A              must be of type real.

KIND (optional)    must be a scalar integer initialization expression.

### Result Type and Type Parameter

The result is of type real.  If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

### Result Value

If $|A| < 1$, AINT(A) has the value 0; if $A \geq 1$, AINT(A) has a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

### Examples

```
AINT(2.783) has the value 2.0.
AINT(-2.783) has the value -2.0.
AINT(2.1111111111111_HIGH) and AINT(2.1111111111111,
HIGH) have the value 2.0 with kind HIGH.
```

## ALL(MASK, DIM)

### Optional Argument

DIM

### Description

Determine whether all values are `.TRUE.` in MASK along dimension DIM.

### Class

Transformational function.

### Arguments

MASK

must be of type logical.  It must not be scalar.

DIM (optional)

must be scalar and of type integer with value in 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 Type, Type Parameter, and Shape

The result is of type logical with the same kind type parameter as MASK.  It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n$-1 and of shape $(d_1, d_2, ..., d_{\text{DIM-1}}, d_{\text{DIM+1}}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

**Result Value**

Case 1    The result of `ALL(MASK)` has the value `.TRUE.` if all elements of `MASK` are `.TRUE.` or if `MASK` has size zero, and the result has value `.FALSE.` if any element of `MASK` is `.FALSE.`.

Case 2    If `MASK` has rank one, `ALL(MASK, DIM)` has a value equal to that of `ALL(MASK)`. Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of `ALL(MASK, DIM)` is equal to `ALL(MASK` $(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n))$.

**Examples**

Case 1    The value of `ALL((/ .TRUE., .FALSE., .TRUE. /))` is `.FALSE.`.

`ALL((/ .TRUE._BIT, .TRUE._BIT, .TRUE._BIT /))` is the value `.TRUE._BIT`.

Case 2    If `B` is the array

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

and `C` is the array

$$\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$$

then `ALL(B .NE. C, DIM = 1)` is [`.TRUE.`, `.FALSE.`, `.FALSE.`] and `ALL(B .NE. C, DIM = 2)` is [`.FALSE.`, `.FALSE.`].

# ALLOCATED(ARRAY)

### Description

Indicate whether or not an allocatable array is currently allocated.

### Class

Inquiry function.

### Argument

ARRAY must be an allocatable array.

### Result Type, Type Parameter, and Shape

Default logical scalar.

### Result Value

The result has the value .TRUE. if ARRAY is currently allocated and has the value .FALSE. if ARRAY is not currently allocated.  The result is undefined if the allocation status of the array is undefined.

### Example

If the following statements are processed

```
REAL, ALLOCATABLE :: A(:,:)
ALLOCATE (A(10,10))
PRINT *, ALLOCATED (A)
```

then T is printed.

# AND(I, J)

### Description

Bitwise AND.

### Class

Elemental nonstandard function.

### Arguments

I          must be of type integer.

J          must be of type integer with the same kind type parameter as I.

### Result Type and Type Parameter

Same as I.

### Result Value

The result has the value obtained by performing a bitwise AND on I and J according to the following truth table:

| I | J | AND(I, J) |
|---|---|-----------|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

The model for interpreting an integer value as a sequence of bits is in the section "The Bit Model".

### Example

The following program produces the output " 11 5 1".

`K` is assigned the binary value 0001, which is 1 in decimal.

```
PROGRAM andtest
INTEGER I, J, K
  I = B'1011'
  J = B'0101'
  K = AND(I, J)
  PRINT *, I, J, K
END
```

# ANINT(A, KIND)

### Optional Argument

`KIND`

### Description

Nearest whole number.

### Class

Elemental function.

### Arguments

`A`                must be of type real.

`KIND` (optional)    must be a scalar integer initialization expression.

### Result Type and Type Parameter

The result is of type real. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

### Result Value

If A>0, ANINT(A) has the value AINT(A+0.5); if A ≤ 0, ANINT(A) has the value AINT(A−0.5).

### Examples

ANINT(2.783) has the value 3.0.

ANINT(−2.783) has the value −3.0.

ANINT(2.7837837837837_HIGH) and
ANINT(2.7837837837837,HIGH) have the value 3.0 with kind HIGH.

# ANY(MASK, DIM)

### Optional Argument

DIM

### Description

Determine whether any value is .TRUE. in MASK along dimension DIM.

### Class

Transformational function.

### Arguments

MASK                must be of type logical. It must not be scalar.

DIM (optional)   must be scalar and of type integer with a value in 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 Type, Type Parameter, and Shape

The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank $n$-1 and of shape $(d_1, d_2, ..., d_{\text{DIM-1}}, d_{\text{DIM+1}}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of MASK.

## Result Value

Case 1   The result of ANY(MASK) has the value .TRUE. if any element of MASK is .TRUE. and has the value .FALSE. if no elements are .TRUE. or if MASK has size zero.

Case 2   If MASK has rank one, ANY(MASK,DIM) has a value equal to that of ANY(MASK). Otherwise, the value of element ($s1, s2, ..., s$DIM-1, $s$DIM+1, ..., $sn$) of ANY(MASK, DIM) is equal to ANY(MASK($s1, s2, ..., s$DIM-1, :, $s$DIM+1, ..., $sn$)).

## Examples

Case 1   The value of ANY((/ .TRUE., .FALSE., .TRUE. /)) is .TRUE..

ANY((/ .FALSE._BIT, .FALSE._BIT, .FALSE._BIT /)) is .FALSE._BIT.

Case 2   If B is the array

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

and C is the array

$$\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$$

then `ANY(B .NE. C, DIM = 1)` is [`.TRUE.`, `.FALSE.`, `.TRUE.`] and `ANY(B .NE. C, DIM = 2)` is [`.TRUE.`, `.TRUE.`].

# ASIN(X)

### Description

Arcsine (inverse sine) function in radians.

### Class

Elemental function.

### Argument

X must be of type real. Its value must satisfy the inequality $|X| \leq 1$.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to arcsin(X), expressed in radians. It lies in the range $-\pi/2 \leq$ `ASIN(X)` $\leq \pi/2$.

### Examples

`ASIN(0.84147098)` has the value 1.0.

`ASIN(1.0_HIGH)` has the value 1.5707963267949 with kind `HIGH`.

# ASIND(X)

### Description

Arcsine (inverse sine) function in degrees.

### Class

Elemental nonstandard function.

### Argument

X must be of type real. Its value must satisfy the inequality $|X| \leq 1$.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to arcsin(X), expressed in degrees. It lies in the range $-90 \leq ASIN(X) \leq 90$.

### Examples

ASIND(-1.0) has the value -90.0.

ASIND(0.5) has the 7value 30.0.

# ASINH(X)

### Description

Hyperbolic arcsine of radians.

### Class

Elemental nonstandard function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to the hyperbolic arcsine of X.

### Examples

ASINH(-1.0) has the value -0.88137.

ASINH(180.0) has the value 5.88611.

# ASSOCIATED(POINTER, TARGET)

### Optional Argument

TARGET

### Description

Returns the association status of its pointer argument or indicates the pointer is associated with the target.

### Class

Inquiry function.

### Arguments

POINTER                must be a pointer and may be of any type.  Its
                       pointer association status must not be undefined.

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

### Result Type

The result is scalar of type default logical.

### Result Value

Case 1                 If TARGET is absent, the result is .TRUE. if POINTER is
                       currently associated with a target and .FALSE. if it
                       is not.

Case 2                 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.

Case 3    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 is .FALSE.
otherwise. If either POINTER or TARGET is
disassociated, the result is .FALSE..

**Examples**

Case 1    ASSOCIATED(PTR) is .TRUE. if PTR is currently
associated with a target.

Case 2    ASSOCIATED(PTR, TAR) is .TRUE. if the following
statements have been processed:

```
REAL, TARGET  :: TAR (0:100)
```

```
REAL, POINTER :: PTR(:)
```

```
PTR => TAR
```

The subscript range for both TAR and PTR is 0:100.

If the pointer assignment statement is either

```
PTR => TAR(:)
```

or

```
PTR => TAR(0:100)
```

then ASSOCIATED(PTR, TAR) is still .TRUE., but in
both cases the subscript range for PTR is 1:101.

However, if the pointer assignment statement is

```
PTR => TAR(0:99)
```

then ASSOCIATED(PTR, TAR) is .FALSE., because
TAR(0:99) is not the same as TAR.

Case 3        `ASSOCIATED(PTR1, PTR2)` is `.TRUE.` if the following statements have been processed.

`REAL, POINTER :: PTR1(:), PTR2(:)`

`ALLOCATE(PTR1(0:10))`

`PTR2 => PTR1`

After the execution of either

`NULLIFY(PTR1)`

or

`NULLIFY(PTR2)`

the statement `ASSOCIATED(PTR1, PTR2)` evaluates to `.FALSE.`.

# ATAN(X)

### Description

Arctangent (inverse tangent) function in radians.

### Class

Elemental function.

### Argument

`X` must be of type real.

### Result Type and Type Parameter

Same as `X`.

### Result Value

The result has a value equal to a processor-dependent approximation to arctan(X), expressed in radians, that lies in the range $-\pi/2 \leq$ ATAN(X) $\leq \pi/2$.

### Examples

ATAN(1.5574077) has the value 1.0.

ATAN(2.0_HIGH/3.0) has the value 0.58800260354757 with kind HIGH.

# ATAN2(Y, X)

### Description

Arctangent (inverse tangent) function in radians. The result is the principal value of the argument of the nonzero complex number (X, Y).

### Class

Elemental function.

### Arguments

Y                  must be of type real.

X                  must be of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

### Result Type and Type Parameter

Same as X.

**Result Value**

The result has a value equal to a processor-dependent approximation to the principal value of the argument of the complex number (X, Y), expressed in radians.

The result lies in the range -$\pi \leq$ ATAN2(Y, X) $\leq \pi$ and is equal to a processor-dependent approximation to a value of arctan(Y/X) if X≠0.

If Y>0, the result is positive.  If Y=0, the result is zero if X>0 and the result is $\pi$ if X<0.  If Y<0, the result is negative.  If X=0, the absolute value of the result is $\pi/2$.

**Examples**

ATAN2(1.5574077, 1.0) has the value 1.0.

If Y has the value

$$\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$$

and X has the value

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$

then the value of ATAN2(Y, X) is

$$\begin{bmatrix} \dfrac{3\pi}{4} & \dfrac{\pi}{4} \\ \dfrac{-3\pi}{4} & -\dfrac{\pi}{4} \end{bmatrix}$$

# ATAN2D(Y, X)

### Description

Arctangent (inverse tangent) function in degrees.

### Class

Elemental nonstandard function.

### Arguments

Y          must be of type real.

X          must be of the same type and kind type parameter as Y.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to the principal value of the argument of the complex number (X, Y), expressed in degrees, that lies in the range $-90 < \text{ATAN2D}(Y,X) < 90$.

### Examples

ATAN2D(1.0, 1.0) has the value 45.0.

ATAN2D(1.0, 0.0) has the value 90.0.

ATAN2D(8735.0, 1.0) has the value 89.99344.

# ATAND(X)

### Description

Arctangent (inverse tangent) function in degrees.

### Class

Elemental nonstandard function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to arctan(X), expressed in degrees, that lies in the range $-90 <$ ATAND(X) $< 90$.

### Examples

ATAND(1.0) has the value 45.0.

ATAND(0.0) has the value 0.0.

ATAND(-94373.0) has the value -89.9994.

# ATANH(X)

### Description

Hyperbolic arctangent of radians.

### Class

Elemental nonstandard function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to the hyperbolic arctangent of X.

### Examples

ATANH(0.0) has the value 0.0.

ATANH(-0.77) has the value -1.02033.

ATANH(0.5) has the value 0.549306.

# BADDRESS(X)

### Description

Return the address of X.

### Class

Inquiry nonstandard function.

### Argument

X may be of any type.

### Result Type

The result is of type default integer.

Example.

The following program:

```
PROGRAM batest
  INTEGER X(5), I
  DO I=1, 5
    PRINT *, BADDRESS(X(I))
  END DO
END
```

Could produce this output:

```
2063835808
2063835812
2063835816
2063835820
2063835824
```

# BIT_SIZE(I)

### Description

Returns the number of bits $n$, defined by the model in the section "The Bit Model", for integers with the kind parameter of the argument.

### Class

Inquiry function.

### Argument

`I` must be of type integer.

### Result Type, Type Parameter, and Shape

Scalar integer with the same kind type parameter as `I`.

### Result Value

The result has the value of the number of bits $n$ in the model integer, defined for bit manipulation contexts in the section "The Bit Model", for integers with the kind parameter of the argument.

### Examples

`BIT_SIZE(1)` has the value 32 if $n$ in the model is 32.

# BTEST(I, POS)

### Description

Tests a bit of an integer value.

### Class

Elemental function.

### Argument

I          must be of type integer.

POS       must be of type integer. It must be nonnegative and be less than `BIT_SIZE(I)`.

### Result Type

The result is of type default logical.

### Result Value

The result has the value `.TRUE.` if bit POS of I has the value 1 and has the value `.FALSE.` if bit POS of I has the value 0. The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

### Examples

`BTEST(8, 3)` has the value `.TRUE.`.

`BTEST(8_SHORT, 3)` has the value `.TRUE.`.

If A has the value [1, 2, 3, 4] then the value of `BTEST(A, 2)` is [`.FALSE.`, `.FALSE.`, `.FALSE.`, `.TRUE.`] and the value of `BTEST(2, A)` is [`.TRUE.`, `.FALSE.`, `.FALSE.`, `.FALSE.`].

# CEILING(A)

### Description

Returns the least integer greater than or equal to its argument.

### Class

Elemental function.

### Argument

A must be of type real.

### Result Type and Type Parameter

Default integer.

### Result Value

The result has a value equal to the least integer greater than or equal to A. The result is undefined if the processor cannot represent this value in the default integer type.

### Examples

CEILING(3.7) has the value 4.

CEILING(-3.7) has the value –3.

CEILING(20.0_HIGH/3) has the value 7.

# CHAR(I, KIND)

## Optional Argument

KIND

## Description

Returns the character in a given position of the processor collating sequence associated with the specified kind type parameter.  It is the inverse of the function ICHAR.

## Class

Elemental function.

## Arguments

I                must be of type integer with a value in the range $0 \leq$ I $\leq$ *n*-1, where *n* is the number of characters in the collating sequence associated with the specified kind type parameter.

KIND (optional)    must be a scalar integer initialization expression.

## Result Type and Type Parameters

Character of length one.  If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default character type.

**Result Value**

The result is the character in position I of the collating sequence associated with the specified kind type parameter.

ICHAR(CHAR(I, KIND(C))) must have the value I for $0 \leq I \leq n\text{-}1$ and CHAR(ICHAR(C), KIND(C)) must have the value C for any character C capable of representation in the processor.

**Example.**

CHAR(88) is 'X' on a processor using the ASCII collating sequence.

# CMPLX(X, Y, KIND)

**Optional Arguments**

Y, KIND

**Description**

Convert to complex type.

**Class**

Elemental function.

**Arguments**

| | |
|---|---|
| X | must be of type integer, real, or complex. |
| Y (optional) | must be of type integer or real.  It must not be present if X is of type complex. |
| KIND (optional) | must be a scalar integer initialization expression. |

### Result Type and Type Parameter

The result is of type complex.  If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default real type.

### Result Value

If Y is absent and X is not complex, it is as if Y were present with the value zero.

If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X).

CMPLX(X,Y,KIND) has the complex value whose real part is REAL(X,KIND) and whose imaginary part is REAL(Y,KIND).

### Examples

CMPLX(−3) is −3.0 + 0*i*.

CMPLX((4.1,0.0), KIND=HIGH), CMPLX((4.1,0), KIND=HIGH), and CMPLX(4.1, KIND=HIGH) are each 4.1 + 0.0*i* with kind HIGH.

# CONJG(Z)

### Description

Conjugate of a complex number.

### Class

Elemental function.

### Argument

Z must be of type complex.

### Result Type and Type Parameter

Same as Z.

### Result Value

If Z has the value $(x, y)$, the result has the value $(x, -y)$.

### Examples

```
CONJG((2.0, 3.0) is (2.0, - 3.0).
CONJG((0., -4.1_HIGH)) is 0 + 4.1i with kind HIGH.
```

# COS(X)

### Description

Cosine function in radians.

### Class

Elemental function.

### Argument

X must be of type real or complex.

### Result Type and Type Parameter

Same as X.

**Result Value**

The result has a value equal to a processor-dependent approximation to cos(X).  If X is of type real, it is regarded as a value in radians.  If X is of type complex, its real part is regarded as a value in radians.

**Examples**

COS(1.0) has the value 0.54030231.

COS((1.0_HIGH, 1.0)) has the value $0.83373002513115 - 0.98889770576287i$ with kind HIGH.

# COSD(X)

### Description

Cosine  function that accepts input in degrees.

### Class

Elemental nonstandard function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to cos(X).

### Examples

COSD(0.0) has the value 1.0.

COSD(60.0) has the value 0.5.

# COSH(X)

## Description

Hyperbolic cosine function.

## Class

Elemental function.

## Argument

X must be of type real.

## Result Type and Type Parameter

Same as X.

## Result Value

The result has a value equal to a processor-dependent approximation to cosh(X).

### Examples

COSH(1.0) has the value 1.5430806.

COSH(0.1_HIGH) has the value 1.0050041680558 with kind HIGH.

# COUNT(MASK, DIM)

### Optional Argument

DIM

### Description

Count the number of .TRUE. elements of MASK along dimension DIM.

**Class.** Transformational function.

### Argument

MASK            must be of type logical.  It must not be scalar.

DIM (optional)  must be scalar and of type integer with a value in 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 Type, Type Parameter, and Shape

The result is of type default integer.  It is scalar if DIM is absent or MASK 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 MASK.

**Result Value**

Case 1    The result of COUNT(MASK) has a value equal to the number of .TRUE. elements of MASK or has the value zero if MASK has size zero.

Case 2    If MASK has rank one, COUNT(MASK, DIM) has a value equal to that of COUNT(MASK). Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of COUNT(MASK, DIM) is equal to COUNT(MASK($s_1$, $s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$)).

**Examples**

Case 1    The value of COUNT((/ .TRUE., .FALSE., .TRUE. /)) is 2.

Case 2    If B is the array

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

and C is the array

$$\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$$

then COUNT(B .NE. C, DIM = 1) is [2, 0, 1] and COUNT(B .NE. C, DIM = 2) is [1, 2].

# CPU_TIME(TIME)

### Description

CPU_TIME returns a processor dependent time in seconds. To get elapsed CPU_TIME, you must call the intrinsic twice, once to get the start time, and again to get a finish time, and then subtract start from finish.

### Class

Subroutine

### Argument

TIME must be scalar and of type real.

As an extension Intel Fortran allows TIME to be of type REAL*8, REAL*16, or DOUBLE-PRECISION.

### Result Type and Type Parameter

Same as TIME.

### Example

```
  PROGRAM TIMEIT
REAL STARTTIME/0.0/,STOPTIME/0.0/
A = 1.2
CALL CPU_TIME(STARTTIME)
  DO I = 1,100000
     B = CCOS(CMPLX(a,0.0)) * CSIN(CMPLX(a,0.0))
  ENDDO
  CALL CPU_TIME(STOPTIME)
  PRINT *,'CPU TIME WAS ',STOPTIME - STARTTIME
  END
```

---

# CSHIFT(ARRAY, SHIFT, DIM)

---

## Optional Argument

DIM

## Description

Perform a circular shift on an array expression of rank one, or perform circular shifts on all the complete rank one sections along a given dimension of an array expression of rank two or greater.

Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions (positive for left shifts, negative for right shifts).

## Class

Transformational function.

## Arguments

ARRAY          may be of any type. It must not be scalar.

SHIFT          must be of type integer and must be scalar if ARRAY has 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.

DIM (optional)  must be a scalar and of type integer with a value in the range $1 \leq DIM \leq n$, where $n$ is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

## Result Type, Type Parameter, and Shape

The result is of the type and type parameters of ARRAY, and has the shape of ARRAY.

**Result Value**

Case 1
If ARRAY has rank one, element $i$ of the result is
ARRAY(1 + MODULO($i$ + SHIFT – 1,
SIZE(ARRAY))).

Case 2
If ARRAY has rank greater than one, section (s1, s2,
..., sDIM-1, :, sDIM+1, ...., sn) of the result
has a value equal to CSHIFT(ARRAY($s_1$, $s_2$, ...,
$s_{DIM-1}$, :, $s_{DIM+1}$, ...., $s_n$), $sh$, 1), where $sh$
is SHIFT or SHIFT($s_1$, $s_2$, ..., $s_{DIM-1}$, $s_{DIM+1}$,
..., $s_n$).

**Examples**

Case 1
If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V
circularly to the left by two positions is achieved by
CSHIFT(V, SHIFT = 2) which has the value
[3, 4, 5, 6, 1, 2].

CSHIFT(V, SHIFT = –2) achieves a circular shift to
the right by two positions and has the value
[5, 6, 1, 2, 3, 4].

Case 2
The rows of an array of rank two may all be shifted by
the same amount or by different amounts. If M is the
array

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

then the value of CSHIFT(M, SHIFT = –1, DIM = 2) is

$$\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$$

and the value of CSHIFT(M, SHIFT =
(/ –1, 1, 0 /), DIM = 2) is

$$\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$$

# DATE_AND_TIME(DATE, TIME, ZONE, VALUES)

## Optional Arguments

DATE, TIME, ZONE, VALUES

## Description

Returns data on the real-time clock and date in a form compatible with the representations defined in ISO 8601:1988 ("Data elements and interchange formats — Information interchange — Representation of dates and times").

## Class

Subroutine.

## Arguments

DATE (optional)    must be scalar and of type default character, and must be of length at least 8 in order to contain the complete value. It is an INTENT(OUT) argument. Its leftmost 8 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 (optional)    must be scalar and of type default character, and must be of length at least 10 in order to contain the complete value. It is an INTENT(OUT) argument. Its leftmost 10 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 (optional)     must be scalar and of type default character, and must be of length at least 5 in order to contain the complete value. It is an INTENT(OUT) argument. Its leftmost 5 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) in hours and parts of an hour expressed in minutes, respectively. If there is no clock available, they are set to blank.

VALUES (optional)   must be of type default integer and of rank one. It is an INTENT(OUT) argument. Its size must be at least 8. 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 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.

The HUGE intrinsic function is described in the section "HUGE(X)".

### Example

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 10) BIG_BEN (3)
CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2),  &
                    BIG_BEN (3), DATE_TIME)
```

if called in Geneva, Switzerland on 1985 April 12 at 15:27:35.5 would have assigned the value "19850412bb" to BIG_BEN(1), the value "152735.500" to BIG_BEN(2), and the value "+0100bbbbbb" to BIG_BEN(3), and the following values to DATE_TIME: 1985, 4, 12, 60, 15, 27, 35, 500.

Note that UTC is defined by CCIR Recommendation 460-2 (and is also known as Greenwich Mean Time).

# DBLE(A)

### Description

Convert to double precision real type.

### Class

Elemental function.

### Argument

A must be of type integer, real, or complex.

### Result Type and Type Parameter

Double precision real.

### Result Value

Case 1            If A is of type double precision real, DBLE(A) = A.

Case 2          If A is of type integer or real, the result is as much precision of the significant part of A as a double precision real datum can contain.

Case 3          If A is of type complex, the result is as much precision of the significant part of the real part of A as a double precision real datum can contain.

### Examples

DBLE(-.3) is –0.3 of type double precision real.

DBLE(1.0_HIGH/3) is 0.33333333333333 of type double precision real.

# DFLOAT(A)

### Description

Convert to double precision type.

### Class

Elemental nonstandard function.

### Argument

A must be of type integer.

### Result Type and Type Parameter

Double precision.

### Example

DFLOAT(56) is 56.0 of type double precision.

# DIGITS(X)

### Description

Returns the number of significant digits in the model representing numbers of the same type and kind type parameter as the argument.

### Class

Inquiry function.

### Argument

X must be of type integer or real. It may be scalar or array valued.

### Result Type, Type Parameter, and Shape

Default integer scalar.

### Result Value

The result has the value $q$ if X is of type integer and $p$ if X is of type real, where $q$ and $p$ are as defined in the section "Data Representation Models" for the model representing numbers of the same type and kind type parameter as X.

### Example

DIGITS(X) has the value 24 for real X whose model is described in the section "The Real Number System Model".

# DIM(X, Y)

### Description

The difference X–Y if it is positive; otherwise zero.

### Class

Elemental function.

### Argument

X          must be of type integer or real.

Y          must be of the same type and kind type parameter as X.

### Result Type and Type Parameter

Same as X.

### Result Value

The value of the result is X–Y if X > Y and zero otherwise.

### Examples

DIM(5, 3) has the value 2. DIM(-3.0, 2.0) has the value 0.0.

# DNUM(I)

### Description

Convert to double precision.

### Class

Elemental nonstandard function.

### Argument

`I` must be of type character.

### Result Type

Double precision.

### Examples

`DNUM("3.14159")` is 3.14159 of type double precision.

The following code sets `x` to 311.0:

```
CHARACTER(3) i
DOUBLE PRECISION x
i = "311"
x = DNUM(I)
```

# DOT_PRODUCT(VECTOR_A, VECTOR_B)

### Description

Performs dot-product multiplication of numeric or logical vectors.

### Class

Transformational function.

### Argument

VECTOR_A        must be of numeric type (integer, real, or complex) or of logical type.  It must be array valued and of rank one.

VECTOR_B        must be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical.  It must be array valued and of rank one.  It must be of the same size as VECTOR_A.

### Result Type, Type Parameter, and Shape

The result is scalar.

If the arguments are of numeric type, the type and kind type parameter of the result are those of the expression VECTOR_A * VECTOR_B determined by the types of the arguments.

If the arguments are of type logical, the result is of type logical with the kind type parameter of the expression VECTOR_A .AND. VECTOR_B.

**Result Value**

Case 1          If VECTOR_A is of type integer or real, the result has the
                value SUM(VECTOR_A*VECTOR_B). If the vectors have
                size zero, the result has the value zero.

Case 2          If VECTOR_A is of type complex, the result has the value
                SUM(CONJG(VECTOR_A)*VECTOR_B). If the vectors
                have size zero, the result has the value zero.

Case 3          If VECTOR_A is of type logical, the result has the value
                ANY(VECTOR_A .AND. VECTOR_B). If the vectors
                have size zero, the result has the value .FALSE..

**Examples**

Case 1          DOT_PRODUCT( (/ 1, 2, 3 /), (/ 2, 3,
                4 /) ) has the value 20.

Case 2          DOT_PRODUCT( (/ (1.0, 2.0), (2.0,
                3.0) /), (/ (1.0, 1.0), (1.0, 4.0) /) ) has
                the value $17 + 4i$.

Case 3          DOT_PRODUCT( (/ .TRUE., .FALSE. /),
                (/ .TRUE., .TRUE. /) ) has the value .TRUE..

# DPROD(X, Y)

**Description**

Double precision real product.

**Class**

Elemental function.

**Argument**

X                    must be of type default real.

Y                    must be of type default real.

**Result Type and Type Parameters**

Double precision real.

**Result Value**

The result has a value equal to a processor-dependent approximation to the product of X and Y.

**Example**

DPROD(–3.0, 2.0) has the value –6.0 of type double precision real.

# DREAL(A)

**Description**

Convert to double precision.

**Class**

Elemental nonstandard function.

**Argument**

A must be of type integer, real, or complex.

**Result**

Double precision.

### Examples

DREAL(91) is 91.0 of type double precision.

The following code sets x to 45.34:

```
COMPLEX p
DOUBLE PRECISION x
p = (45.34, 1.0)
x = DREAL(p)
```

# DSIGN

*Returns the absolute value multiplied by
the sign of the second argument*

### Description

This funciton performs a sign transfer by returning the absolute value of the
first argument multiplied by the sign of the second argument.

### Class

Non-standard elemental function.

Prototype

### Arguments

A1              number whose absolute value is the magnitude of the
                result

A2              number whose sign is the sign of the result

### Result

|A1| if A2>=0 or

-|A1| if b<0

If A2 is zero,

if the process cannot distinguish between $+0$ and $-0$,

the result is $|\mathtt{A1}|$

otherwise, for $-0$, the value is $-|\mathtt{A1}|$

for $+0$, the value is $|\mathtt{A1}|$

# EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)

## Optional Argument

BOUNDARY**,** DIM

## Description

Perform an end-off shift on an array expression of rank one or perform end-off shifts on all the complete rank-one sections along a given dimension of an array expression of rank two or greater.

Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other end.

Different sections may have different boundary values and may be shifted by different amounts and in different directions (positive for left shifts, negative for right shifts).

## Class

Transformational function.

## Argument

ARRAY          may be of any type. It must not be scalar.

SHIFT          must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank n-1 and of shape $(d_1, d_2, \ldots, d_{\mathtt{DIM}-1}, d_{\mathtt{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

BOUNDARY (optional)     must be of the same type and type parameters as
                        ARRAY and must be scalar if ARRAY has rank one;
                        otherwise, it must be either scalar or of rank $n$-1
                        and of shape ($d_1$, $d_2$, ..., $d_{DIM-1}$, $d_{DIM+1}$,
                        ..., $d_n$). BOUNDARY may be omitted for the
                        data types in the following table and, in this case,
                        it is as if it were present with the scalar value
                        shown.

| Type of ARRAY | Value of BOUNDARY |
|---------------|-------------------|
| Integer | 0 |
| Real | 0.0 |
| Complex | (0.0, 0.0) |
| Logical | .FALSE. |
| Character (*len*) | *len* blanks |

DIM (optional)      must be scalar and of type 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 1.

## Result Type, Type Parameter, and Shape

The result has the type, type parameters, and shape of ARRAY.

## Result Value

Element ($s_1$, $s_1$, ..., $s_n$) of the result has the value ARRAY ($s_1$, $s_2$,
..., $s_{DIM-1}$, $s_{DIM}$ + $sh$, $s_{DIM+1}$, ..., $s_n$) where $sh$ is SHIFT or
SHIFT($s_1$, $s_2$, ..., $s_{DIM-1}$, $s_{DIM+1}$, ..., $s_n$) provided the
inequality LBOUND(ARRAY, DIM) $\le s_{DIM}$ + $sh \le$ UBOUND(ARRAY, DIM)
holds and is otherwise BOUNDARY or BOUNDARY($s_1$, $s_2$, ..., $s_{DIM-1}$,
$s_{DIM+1}$, ..., $s_n$).

## Examples

Case 1          If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V
                end-off to the left by 3 positions is achieved by
                EOSHIFT(V, SHIFT = 3) which has the value
                [4, 5, 6, 0, 0, 0].

`EOSHIFT(V, SHIFT = -2, BOUNDARY = 99)` achieves an end-off shift to the right by 2 positions with the boundary value of 99 and has the value [99, 99, 1, 2, 3, 4].

Case 2    The rows of an array of rank two may all be shifted by the same amount or by different amounts and the boundary elements can be the same or different. If `M` is the array

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$$

then the value of `EOSHIFT(M, SHIFT = -1, BOUNDARY = '*', DIM = 2)` is

$$\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix}$$

and the value of `EOSHIFT(M, SHIFT = (/ -1, 1, 0 /), BOUNDARY = (/ '*', '/', '?' /), DIM = 2)` is

$$\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}$$

# EPSILON(X)

## Description

Returns a positive model number that is almost negligible compared to unity in the model representing numbers of the same type and kind type parameter as the argument.

### Class

Inquiry function.

### Argument

X must be of type real.  It may be scalar or array valued.

### Result Type, Type Parameter, and Shape

Scalar of the same type and kind type parameter as X.

### Result Value

The result has the value $b^{1-p}$ where $b$ and $p$ are as defined in the section "The Real Number System Model" for the model representing numbers of the same type and kind type parameter as X.

### Examples

EPSILON(X) has the value $2^{-23}$ for real X whose model is described in the section "The Real Number System Model".

EPSILON(Y), where Y has kind parameter HIGH, would be $2^{-47}$ if $p$ is 48 for the model of kind HIGH.

# EXP(X)

### Description

Exponential.

### Class

Elemental function.

### Argument

X must be of type real or complex.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to $e^x$. If X is of type complex, its imaginary part is regarded as a value in radians.

### Examples

EXP(1.0) has the value 2.7182818.

EXP(2.0_HIGH/3.0) has the value 1.9477340410547 with kind HIGH.

# EXPONENT(X)

### Description

Returns the exponent part of the argument when represented as a model number.

### Class

Elemental function.

### Argument

X must be of type real.

### Result Type

Default integer.

### Result Value

The result has a value equal to the exponent *e* of the model representation (see the section "The Real Number System Model") for the value of X, provided X is nonzero and *e* is within the range for default integers. The result is undefined if the processor cannot represent *e* in the default integer type. EXPONENT(X) has the value zero if X is zero.

### Examples

EXPONENT(1.0) has the value 1 and EXPONENT(4.1) has the value 3 for reals, whose model is described in the section "The Real Number System Model".

# FLOOR(A)

### Description

Returns the greatest integer less than or equal to its argument.

### Class

Elemental function.

### Argument

A must be of type real.

### Result Type and Type Parameter

Default integer.

### Result Value

The result has a value equal to the greatest integer less than or equal to A. The result is undefined if the processor cannot represent this value in the default integer type.

### Examples

FLOOR(3.7) has the value 3.

FLOOR(-3.7) has the value –4.

FLOOR(10.0_HIGH/3) has the value 3.

# FRACTION(X)

### Description

Returns the fractional part of the model representation of the argument value.

### Class

Elemental function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has the value $X \times b^{-e}$, where $b$ and $e$ are as defined in the section "The Real Number System Model".  If X has the value zero, the result has the value zero.

### Example

FRACTION(3.0) has the value 0.75 for reals, whose model is described in "The Real Number System Model".

# HFIX(A)

### Description

Convert to `INTEGER(2)` type.

### Class

Elemental nonstandard function.

### Argument

`A` must be of type integer, real, double precision, or complex.

### Result

`INTEGER(2)` type.

### Examples

`HFIX(9.897)` is 9 of type `INTEGER(2)`.

`HFIX(9.125)` is 9 of type `INTEGER(2)`.

The following code sets `b` to 34:

```
INTEGER(2) b
COMPLEX p
p = (34.5, 1.0)
b = HFIX(p)
```

# HUGE(X)

### Description

Returns the largest number in the model representing numbers of the same type and kind type parameter as the argument.

### Class

Inquiry function.

### Argument

X must be of type integer or real. It may be scalar or array valued.

**Result Type, Type Parameter, and Shape.** Scalar of the same type and kind type parameter as X.

### Result Value

The result has the value $r^q$ - 1 if X is of type integer and

$$(1 - b^{-p})b^{e_{max}}$$

if X is of type real, where $r$, $q$, $b$, $p$, and $e_{max}$ are as defined in the section "The Real Number System Model".

### Example

HUGE(X) has the value $(1 - 2^{-24}) \times 2^{127}$ for real X, whose model is described in "The Real Number System Model".

# IABS(A)

*Returns the absolute value of an integer
expression*

### Description

`IABS` returns the absolute value of an `INTEGER*2` expression.

### Class

Elemental function.

### Argument

`A` must be of type INTEGER*2 value or expression.

### Result Value

If `A` is positive or zero, the value of `A` is returned. If `A` is less than zero, the opposite positive value of `A` is returned.

### Output

Absolute value of `A`.

# IACHAR(C)

### Description

Returns the position of a character in the ASCII collating sequence.

### Class

Elemental function.

### Argument

C must be of type default character and of length one.

### Result Type and Type Parameter

Default integer.

### Result Value

If C is in the collating sequence defined by the codes specified in ISO 646:1983 ("Information technology — ISO 7-bit coded character set for information interchange"), the result is the position of C in that sequence and satisfies the inequality ($0 \leq$ IACHAR(C) $\leq 127$).

A processor-dependent value is returned if C is not in the ASCII collating sequence. The results are consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE(C, D) is .TRUE., IACHAR(C) .LE. IACHAR(D) is .TRUE. where C and D are any two characters representable by the processor.

### Examples

IACHAR('X') has the value 88.

IACHAR('*') has the value 42.

# IADDR(X)

### Description

Return the address of X.

# IAND(I, J)

### Description

Performs a bitwise logical AND.

### Class

Elemental function.

### Argument

| | |
|---|---|
| I | must be of type integer. |
| J | must be of type integer with the same kind type parameter as I. |

### Result Type and Type Parameter

Same as I.

### Result Value

The result has the value obtained by combining `I` and `J` bit-by-bit according to the following truth table:

| I | J | IAND(I, J) |
|---|---|-----------|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

The model for the interpretation of an integer value as a sequence of bits is in "The Bit Model".

### Examples

`IAND(1, 3)` has the value 1.

`IAND(2_SHORT, 10_SHORT)` is 2 with kind `SHORT`.

# IBCLR(I, POS)

### Description

Clears a bit to zero.

### Class

Elemental function.

### Argument

| | |
|---|---|
| `I` | must be of type integer. |
| `POS` | must be of type integer. It must be nonnegative and less than `BIT_SIZE(I)`. |

**Result Type and Type Parameter**

Same as I.

**Result Value**

The result has the value of the sequence of bits of I, except that bit POS of I is set to zero. The model for the interpretation of an integer value as a sequence of bits is in "The Bit Model".

**Examples**

IBCLR(14, 1) has the result 12.

If V has the value (1, 2, 3, 4), the value of IBCLR(POS = V, I = 31) is [29, 27, 23, 15].

The value of IBCLR((/ 15_SHORT, 31_SHORT, 7_SHORT /), 3) is [7, 23, 7] with kind SHORT.

# IBITS(I, POS, LEN)

**Description**

Extracts a sequence of bits.

**Class**

Elemental function.

**Argument**

| | |
|---|---|
| I | must be of type integer. |
| POS | must be of type integer. It must be nonnegative and POS + LEN must be less than or equal to BIT_SIZE(I). |
| LEN | must be of type integer and nonnegative. |

### Result Type and Type Parameter

Same as I.

### Result Value

The result has the value of the sequence of LEN bits in I beginning at bit POS, right-adjusted and with all other bits zero. The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

### Examples

IBITS(14, 1, 3) has the value 7.

The value of IBITS((/ 15_SHORT, 31_SHORT, 7_SHORT /), 2_SHORT, 3_SHORT) is [3, 7, 1] with kind SHORT.

# IBSET(I, POS)

### Description

Sets a bit to one.

### Class

Elemental function.

### Argument

I               must be of type integer.

POS             must be of type integer. It must be nonnegative and less
                than BIT_SIZE(I).

### Result Type and Type Parameter

Same as I.

**Result Value**

The result has the value of the sequence of bits of I, except that bit POS of I is set to one. The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

**Examples**

IBSET(12, 1) has the value 14.

If V has the value [1, 2, 3, 4], the value of IBSET(POS = V, I = 0) is [2, 4, 8, 16].

The value of IBSET((/ 15_SHORT, 31_SHORT, 7_SHORT /), 3) is [15, 31, 15] with kind SHORT.

# ICHAR(C)

**Description**

Returns the position of a character in the processor collating sequence associated with the kind type parameter of the character.

**Class**

Elemental function.

**Argument**

C must be of type character and of length one. Its value must be that of a character capable of representation in the processor.

**Result Type and Type Parameter**

Default integer.

**Result Value**

The result is the position of C in the processor collating sequence associated with the kind type parameter of C and is in the range $0 \le \text{IACHAR(C)} \le n\text{-}1$, where $n$ is the number of characters in the collating sequence.

For any characters C and D capable of representation in the processor, C.LE.D is .TRUE. if and only if ICHAR(C) .LE. ICHAR(D) is .TRUE., and C.EQ.D is .TRUE. if and only if ICHAR(C).EQ. ICHAR(D) is .TRUE..

**Examples**

ICHAR('X') has the value 88 on a processor using the ASCII collating sequence for the default character type.

ICHAR('*') has the value 42 on such a processor.

# IDIM(X, Y)

## Description

Integer positive difference.

## Class

Nonstandard function.

## Argument

X               must be of type integer.

Y               must be of type integer with the same kind type parameter as X.

## Result Type and Type Parameter

Integer of same kind type parameter as X.

**Result Value**

If X > Y, IDIM(X, Y) is X-Y. If X ≤ Y, IDIM(X, Y) is zero.

**Examples**

IDIM(89, 12) is 77.

IDIM(56, 59) is 0.

# IEOR(I, J)

**Description**

Performs a bitwise exclusive OR.

**Class**

Elemental function.

**Argument**

I               must be of type integer.

J               must be of type integer with the same kind type
                parameter as I.

**Result Type and Type Parameter**

Same as I.

**Result Value**

The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

| I | J | IEOR(I, J) |
|---|---|------------|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

**Examples**

IEOR(1, 3) has the value 2.

IEOR( (/ 3_SHORT, 10_SHORT /), 2_SHORT) is [1, 8] with kind SHORT.

# IJINT(A)

**Description**

Convert to INTEGER(2) type.

**Class**

Elemental nonstandard function.

**Argument**

A must be of type INTEGER(4).

**Result**

INTEGER(2) type.

**Example**

IJINT(32) is 32 of type INTEGER(2).

# IMAG(A)

**Description**

Imaginary part of complex number.

**Class**

Elemental nonstandard function.

**Argument**

A must be of type complex or double complex.

**Result**

Real if A is complex.  Double precision if A is double complex.

**Example**

The following code sets x to 2.0:

```
COMPLEX p
REAL x
p = (39.61, 2.0)
x = IMAG(p)
```

# INDEX(STRING, SUBSTRING, BACK)

**Optional Argument**

BACK

**Description**

Returns the starting position of a substring within a string.

**Class**

Elemental function.

**Argument**

| | |
|---|---|
| STRING | must be of type character. |
| SUBSTRING | must be of type character with the same kind type parameter as STRING. |
| BACK (optional) | must be of type logical. |

**Result Type and Type Parameter**

Default integer.

**Result Value**

Case 1　　　　If BACK is absent or present with the value .FALSE., the result is the minimum positive value of I such that STRING(I : I + LEN(SUBSTRING) – 1) = SUBSTRING or zero if there is no such value.

Zero is returned if LEN(STRING) < LEN(SUBSTRING) and one is returned if LEN(SUBSTRING) = 0.

Case 2    If `BACK` is present with the value `.TRUE.`, the result is the maximum value of `I` less than or equal to `LEN(STRING) − LEN(SUBSTRING) + 1` such that `STRING(I : I + LEN(SUBSTRING) − 1) = SUBSTRING` or zero if there is no such value.

Zero is returned if `LEN(STRING) < LEN(SUBSTRING)` and `LEN(STRING) + 1` is returned if `LEN(SUBSTRING) = 0`.

### Examples

`INDEX('FORTRAN', 'R')` has the value 3.

`INDEX('FORTRAN', 'R', BACK = .TRUE.)` has the value 5.

`INDEX("XXX", "")` has the value 1.

`INDEX("XXX", "", BACK=.TRUE.)` has the value 4.

# INT(A, KIND)

### Optional Argument

`KIND`

### Description

Convert to integer type.

### Class

Elemental function.

### Argument

`A`                    must be of type integer, real, or complex.

`KIND` (optional)    must be a scalar integer initialization expression.

### Result Type and Type Parameter

Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default integer type.

### Result Value

| | |
|---|---|
| Case 1 | If A is of type integer, INT(A) = A. |
| Case 2 | If A is of type real, there are two cases: if $|A| < 1$, INT(A) has the value 0; if $|A| \geq 1$, INT(A) is the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A. |
| Case 3 | If A is of type complex, INT(A) is the value obtained by applying the above rules (for reals) to the real part of A. The result is undefined if the processor cannot represent the result in the specified integer type. |

### Examples

INT(-3.7) has the value –3.

INT(9.1_HIGH/4.0_HIGH, SHORT) is 2 with kind SHORT.

# INT1(A)

### Description

Convert to INTEGER(1) type.

### Class

Elemental nonstandard function.

**Argument**

A must be of type integer, real, or complex.

**Result**

INTEGER(1) type. If A is complex, INT1(A) is equal to the truncated real portion of A.

**Example**

INT1(6.23) is 6 of type INTEGER(1).

# INT2(A)

**Description**

Convert to INTEGER(2) type.

**Class**

Elemental nonstandard function.

**Argument**

A must be of type integer, real, or complex.

**Result**

INTEGER(2) type. If A is complex, INT2(A) is equal to the truncated real portion of A.

**Example**

INT2(212.4545) is 212 of type INTEGER(2).

# INT4(A)

### Description

Convert to `INTEGER(4)` type.

### Class

Elemental nonstandard function.

### Argument

`A` must be of type integer, real, or complex.

### Result

`INTEGER(4)` type.  If `A` is complex, `INT4(A)` is equal to the truncated real portion of `A`.

### Example

`INT4(1988.74)` is 1988 of type `INTEGER(4).`

# INT8(A)

### Description

Convert to `INTEGER(8)` type.

### Class

Elemental nonstandard function.

### Argument

`A` must be of type integer, real, or complex.

### Result

`INTEGER(8)` type.  If `A` is complex, `INT8(A)` is equal to the truncated real portion of `A`.

### Example

`INT8(14.14)` is 14 of type `INTEGER(8)`.

# INUM(I)

### Description

Convert character to `INTEGER(2)` type.

### Class

Elemental nonstandard function.

### Argument

`I` must be of type character.

### Result

`INTEGER(2)` type.

### Example

`INUM("451.92")` is 451 of type `INTEGER(2)`.

# IOR(I, J)

### Description

Performs a bitwise inclusive OR.

### Class

Elemental function.

### Argument

I       must be of type integer.

J       must be of type integer with the same kind type parameter as I.

### Result Type and Type Parameter

Same as I.

### Result Value

The result has the value obtained by combining I and J bit-by-bit according to the following truth table:

| I | J | IOR(I, J) |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

# IQINT(A)

### Description

Convert to integer type.

### Class

Elemental nonstandard function.

### Argument

A must be of type `REAL(16)`.

### Result

Integer type.

### Examples

`IQINT(9416.39)` is 9416.

# ISHFT(I, SHIFT)

### Description

Performs a logical shift.

### Class

Elemental function.

**Argument**

| | |
|---|---|
| I | must be of type integer. |
| SHIFT | must be of type integer. The absolute value of SHIFT must be less than or equal to BIT_SIZE(I). |

### Result Type and Type Parameter

Same as I.

### Result Value

The result has the value obtained by shifting the bits of I by SHIFT positions.

If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

### Examples

ISHFT(3, 1) has the value 6.

ISHFT(3, −1) has the value 1.

# ISHFTC(I, SHIFT, SIZE)

### Optional Argument

SIZE

### Description

Performs a circular shift of the rightmost bits.

**Class**

Elemental function.

**Argument**

| | |
|---|---|
| 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. |
| SIZE (optional) | must be of type integer. The value of SIZE must be positive and must not exceed BIT_SIZE(I). If SIZE is absent, it is as if it were present with the value of BIT_SIZE(I). |

**Result Type and Type Parameter**

Same as I.

**Result Value**

The result has the value obtained by shifting the SIZE rightmost bits of I circularly by SHIFT positions.

If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right; and if SHIFT is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered.

The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

**Example**

ISHFTC(3, 2, 3) has the value 5.

# ISIGN(A, B)

### Description

Absolute value of A times the sign of B.

### Class

Elemental nonstandard function.

### Argument

A            must be of type integer.

B            must be of type integer with the same kind type parameter as A.

### Result Type and Type Parameter

Same as A.

### Result Value

The value of the result is |A| if B $\geq$ 0 and -|A| if B < 0.

### Examples

```
ISIGN(-3, 0) is 3.
ISIGN(12, -9) is -12.
```

# ISNAN(X)

### Description

Determine if a value is NaN (not a number).

### Class

Elemental nonstandard function.

### Argument

X must be of type real.

### Result Type

Logical.

### Examples

```
ISNAN(45.4) is .FALSE..
ISNAN(ACOSH(0.0)) is .TRUE..
```

# IXOR(I, J)

### Description

Exclusive OR.

### Class

Elemental nonstandard function.

## Argument

| | |
|---|---|
| I | must be of type integer. |
| J | must be of type integer with the same kind type parameter as I. |

## Result Type and Type Parameter

Same as I.

## Result Value

The result has the value obtained by performing an exclusive OR on I and J bit-by-bit according to the truth table that follows.

| I | J | IXOR(I, J) |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

The model for interpreting an integer value as a sequence of bits is in the section "The Bit Model".

## Example

IXOR(12, 7) is 11.  (Binary 1100 exclusive OR with binary 0111 is binary 1011.)

# JNUM(I)

### Description

Convert character to integer type.

### Class

Elemental nonstandard function.

### Argument

I must be of type character.

### Result

Integer type.

### Example

JNUM("46616.725") is 46616.

# KIND(X)

### Description

Returns the value of the kind type parameter of X.

### Class

Inquiry function.

### Argument

X may be of any intrinsic type.

### Result Type, Type Parameter, and Shape

Default integer scalar.

### Result Value

The result has a value equal to the kind type parameter value of X.

### Examples

KIND(0.0) has the kind type parameter value of default real.

KIND(1.0_HIGH) has the value of the named constant HIGH.

## LBOUND(ARRAY, DIM)

### Optional Argument

DIM

### Description

Returns all the lower bounds or a specified lower bound of an array.

### Class

Inquiry function.

### Argument

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

DIM (optional)    must be scalar and of type integer with a value in the
                  range $1 \leq \text{DIM} \leq n$, where *n* is the rank of ARRAY. The
                  corresponding actual argument must not be an optional
                  dummy argument.

## Result Type, Type Parameter, and Shape

The result is of type default integer. It is scalar if DIM is present; otherwise,
the result is an array of rank one and size *n*, where *n* is the rank of ARRAY.

## Result Value

Case 1            For an array section or for an array expression other than
                  a whole array or array structure component,
                  LBOUND(ARRAY, DIM) has the value 1. For a whole
                  array or array structure component, LBOUND(ARRAY,
                  DIM) has the value:

- equal to the lower bound for subscript DIM of
  ARRAY if dimension DIM of ARRAY does not have
  extent zero or if ARRAY is an assumed-size array of
  rank DIM

or

- one (1), otherwise.

Case 2            LBOUND(ARRAY) has a value whose *i*th component is
                  equal to LBOUND(ARRAY, *i*), for *i* = 1, 2, ..., *n*, where *n*
                  is the rank of ARRAY.

## Examples

If the following statements are processed

```
REAL, TARGET :: A (2:3, 7:10)
REAL, POINTER, DIMENSION (:,:) :: B, C, D
B => A
C => A(:,:)
ALLOCATE ( D(-3:3,-7:7) )
LBOUND(A) is [2, 7], LBOUND(A, DIM=2) is 7, LBOUND(B)
is [2,7], LBOUND(C) is [1,1], and LBOUND(D) is
[-3,-7].
```

# LEN(STRING)

### Description

Returns the length of a character entity.

### Class

Inquiry function.

### Argument

STRING must be of type character. It may be scalar or array valued.

### Result Type, Type Parameter, and Shape

Default integer scalar.

### Result Value

The result has a value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is array valued.

### Example

If C and D are declared by the statements

```
CHARACTER (11) C(100)
CHARACTER (LEN=31) D
```

LEN(C) has the value 11, and LEN(D) has the value 31.

# LEN_TRIM(STRING)

### Description

Returns the length of the character argument without counting trailing blank characters.

### Class

Elemental function.

### Argument

STRING must be of type character.

### Result Type and Type Parameter

Default integer.

### Result Value

The result has a value equal to the number of characters remaining after any trailing blanks in STRING are removed. If the argument contains no nonblank characters, the result is zero.

### Examples

LEN_TRIM('ᵇAᵇBᵇᵇ') has the value 4.

LEN_TRIM('ᵇᵇᵇ') has the value 0.

# LGE(STRING_A, STRING_B)

### Description

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

### Class

Elemental function.

### Argument

STRING_A        must be of type default character.

STRING_B        must be of type default character.

### Result Type and Type Parameters

Default logical.

### Result Value

If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor dependent.

The result is .TRUE. if the strings are equal or if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is .FALSE.. Note that the result is .TRUE. if both STRING_A and STRING_B are of zero length.

### Examples

LGE('apple', 'beans') has the value .FALSE..

LGE('apple', 'applesauce') has the value .FALSE..

LGE('Zebra', 'Yak') has the value .TRUE..

# LGT(STRING_A, STRING_B)

### Description

Tests whether a string is lexically greater than another string, based on the ASCII collating sequence.

### Class

Elemental function.

### Argument

STRING_A        must be of type default character.

STRING_B        must be of type default character.

### Result Type and Type Parameters

Default logical.

### Result Value

If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor-dependent.

The result is .TRUE. if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is .FALSE.. Note that the result is .FALSE. if both STRING_A and STRING_B are of zero length.

### Examples

LGT('apple', 'beans') has the value .FALSE..

LGT('apple', 'applesauce') has the value .FALSE..

LGT('Zebra', 'Yak') has the value .TRUE..

# LLE(STRING_A, STRING_B)

### Description

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

### Class

Elemental function.

### Argument

STRING_A         must be of type default character.

STRING_B         must be of type default character.

### Result Type and Type Parameters

Default logical.

### Result Value

If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor dependent.

The result is .TRUE. if the strings are equal or if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is .FALSE.. Note that the result is .TRUE. if both STRING_A and STRING_B are of zero length.

### Examples

LLE('apple', 'beans') has the value .TRUE..

LLE('apple', 'applesauce') has the value .TRUE..

LLE('Zebra', 'Yak') has the value .FALSE..

# LLT(STRING_A, STRING_B)

### Description

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

### Class

Elemental function.

### Argument

STRING_A  must be of type default character.

STRING_B  must be of type default character.

### Result Type and Type Parameters

Default logical.

### Result Value

If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor-dependent.

The result is .TRUE. if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is .FALSE.. Note that the result is .FALSE. if both STRING_A and STRING_B are of zero length.

### Examples

LLT('apple', 'beans') has the value .TRUE..

LLT('apple', 'applesauce') has the value .TRUE..

LLT('Zebra', 'Yak') has the value .FALSE..

# LOC(X)

### Description

Return the address of the argument.

### Class

Inquiry nonstandard function.

For details see .

# LOG(X)

### Description

Natural logarithm.

### Class

Elemental function.

### Argument

X must be of type real or complex. If X is real, its value must be greater than zero. If X is complex, its value must not be zero.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to $\log_e x$. A result of type complex is the principal value with imaginary part $\omega$ in the range $-\pi < \omega \leq \pi$. The imaginary part of the result is $\pi$ only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

### Examples

`LOG(10.0)` has the value 2.3025851.

`LOG((-0.5_HIGH,0))` has the value $-0.69314718055994 + 3.1415926535898i$ with kind `HIGH`.

# LOG10(X)

### Description

Common logarithm.

### Class

Elemental function.

### Argument

`X` must be of type real. The value of `X` must be greater than zero.

### Result Type and Type Parameter

Same as `X`.

**Result Value**

The result has a value equal to a processor-dependent approximation to $\log_{10} x$.

**Examples**

LOG10(10.0) has the value 1.0.

LOG10(10.0E1000_HIGH) has the value 1001.0 with kind HIGH.

# LOGICAL(L, KIND)

**Optional Argument**

KIND

**Description**

Converts between kinds of logical.

**Class**

Elemental function.

**Argument**

L                     must be of type logical.

KIND (optional     must be a scalar integer initialization expression.

**Result Type and Type Parameter**

Logical.  If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default logical.

**Result Value**

The value is that of L.

**Examples**

`LOGICAL(L .OR. .NOT. L)` has the value `.TRUE.` and is of type default logical, regardless of the kind type parameter of the logical variable `L`.

`LOGICAL(L, BIT)` has kind parameter `BIT` and has the same value as `L`.

# LSHFT(I, SHIFT)

### Description

Left shift.

### Class

Elemental nonstandard function.

For details see "LSHFT" on page 89.

# LSHIFT(I, SHIFT)

### Description

Left shift.

### Class

Elemental nonstandard function.

For details see "LSHIFT" on page 90.

# MATMUL(MATRIX_A, MATRIX_B)

### Description

Performs matrix multiplication of numeric or logical matrices.

### Class

Transformational function.

### Argument

MATRIX_A      must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one or two.

MATRIX_B      must be of numeric type if MATRIX_A is of numeric type and of logical type if MATRIX_A is of logical type. It must be array valued and of rank one or two.

If MATRIX_A has rank one, MATRIX_B must have rank two. If MATRIX_B has rank one, MATRIX_A must have rank two. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

### Result Type, Type Parameter, and Shape

If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of MATRIX_A and MATRIX_B.

If the arguments are of type logical, the result is of type logical with the kind type parameter of the arguments.

The shape of the result depends on the shapes of the arguments as follows:

Case 1        If MATRIX_A has shape $[n, m]$ and MATRIX_B has shape $[m, k]$, the result has shape $[n, k]$.

Case 2        If MATRIX_A has shape $[m]$ and MATRIX_B has shape $[m, k]$, the result has shape $[k]$.

Case 3             If `MATRIX_A` has shape [$n$, $m$] and `MATRIX_B` has shape
                   [$m$], the result has shape [$n$].

## Result Value

Case 1             Element ($i$, $j$) of the result has the value
                   `SUM(MATRIX_A(`$i$`,:) * MATRIX_B(:, `$j$`))` if the
                   arguments are of numeric type and has the value
                   `ANY(MATRIX_A(`$i$`, :) .AND. MATRIX_B(:, `$j$`))` if
                   the arguments are of logical type.

Case 2             Element ($j$) of the result has the value
                   `SUM(MATRIX_A(:) * MATRIX_B(:, `$j$`))` if the
                   arguments are of numeric type and has the value
                   `ANY(MATRIX_A(:) .AND. MATRIX_B(:, `$j$`))` if the
                   arguments are of logical type.

Case 3             Element ($i$) of the result has the value
                   `SUM(MATRIX_A(`$i$`, :) * MATRIX_B(:))` if the
                   arguments are of numeric type and has the value
                   `ANY(MATRIX_A(`$i$`, :) .AND. MATRIX_B(:))` if the
                   arguments are of logical type.

## Examples

If A is the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

and B is the matrix

$$\begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$$

and X is the vector [1, 2] and Y is the vector [1, 2, 3].

Case 1             The result of `MATMUL(A, B)` is the matrix-matrix
                   product AB with the value

$$\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$$

| | |
|---|---|
| Case 2 | The result of MATMUL(X, A) is the vector-matrix product XA with the value [5, 8, 11]. |
| Case 3 | The result of MATMUL(A, Y) is the matrix-vector product AY with the value [14, 20]. |

# MAX(A1, A2, A3, ...)

**Optional Arguments**

A3, ...

**Description**

Maximum value.

**Class**

Elemental function.

**Arguments**

The arguments must all have the same type which must be integer or real, and they must all have the same kind type parameter.

**Result Type and Type Parameter**

Same as the arguments.

**Result Value**

The value of the result is that of the largest argument.

**Examples**

```
MAX(-9.0, 7.0, 2.0) has the value 7.0.
MAX(-1.0_HIGH/3, -0.1_HIGH) is -0.1_HIGH.
```

# MAXEXPONENT(X)

**Description**

Returns the maximum exponent in the model representing numbers of the same type and kind type parameter as the argument.

**Class**

Inquiry function.

**Argument**

X must be of type real.  It may be scalar or array valued.

**Result Type, Type Parameter, and Shape**

Default integer scalar.

**Result Value**

The result has the value $e_{max}$, as defined in the section "The Real Number System Model".

**Example**

MAXEXPONENT(X) has the value 128 for real X, whose model is described in the section "The Real Number System Model".

# MAXLOC(ARRAY, MASK)

### Optional Argument

MASK

### Description

Returns the location of the first element of ARRAY having the maximum value of the elements identified by MASK.

### Class

Transformational function.

### Arguments

ARRAY        must be of type integer or real.  It must not be scalar.

MASK (optional)        must be of type logical and must be conformable with ARRAY.

### Result Type, Type Parameter, and Shape

The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

**Result Value**

Case 1    If MASK is absent, the result is a rank-one array whose
element values are the values of the subscripts of an
element of ARRAY whose value equals the maximum
value of all of the elements of ARRAY.

The *i*th subscript returned lies in the range 1 to $e_i$,
where $e_i$ is the extent of the *i*th dimension of ARRAY.

If more than one element has the maximum value, the
element whose subscripts are returned is the first such
element, taken in array element order.  If ARRAY has size
zero, the value of the result is processor-dependent.

Case 2    If MASK is present, the result is a rank-one array whose
element values are the values of the subscripts of an
element of ARRAY, corresponding to a .TRUE. element
of MASK, whose value equals the maximum value of all
such elements of ARRAY.

The *i*th subscript returned lies in the range 1 to $e_i$, where
$e_i$ is the extent of the *i*th dimension of ARRAY.

If more than one such element has the maximum value,
the element whose subscripts are returned is the first
such element taken in array element order.

If there are no such elements (that is, if ARRAY has size
zero or every element of MASK has the value .FALSE.),
the value of the result is processor-dependent.

In both cases, an element of the result is undefined if the processor cannot
represent the value as a default integer.

### Examples

Case 1          The value of MAXLOC((/ 2, 6, 4, 6 /)) is [2].

If the array B is declared

```
INTEGER, DIMENSION(4:7) :: B = (/ 8, 6, 3, 1 /)
```

the value of MAXLOC(B) is [1].

Case 2          If A has the value

$$
\begin{bmatrix}
0 & -5 & 8 & -3 \\
3 & 4 & -1 & 2 \\
1 & 5 & 6 & -4
\end{bmatrix}
$$

then MAXLOC(A, MASK = A .LT. 6) has the value
[3, 2]. Note that this is true even if A has a declared
lower bound other than 1.

# MAXVAL(ARRAY, DIM, MASK)

### Optional Arguments

DIM, MASK

### Description

Maximum value of the elements of ARRAY along dimension DIM that
correspond to the .TRUE. elements of MASK.

### Class

Transformational function.

## Arguments

ARRAY    must be of type integer or real.  It must not be scalar.

DIM (optional)    must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$ where $n$ is the rank of ARRAY.  The corresponding actual argument must not be an optional dummy argument.

MASK (optional)    must be of type logical and must be conformable with ARRAY.

## Result Type, Type Parameter, and Shape

The result is of the same type and kind type parameter as ARRAY.

It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n$-1 and of shape $(d_1, d_2, \ldots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

## Result Value

Case 1    The result of MAXVAL(ARRAY) has a value equal to the maximum value of all the elements of ARRAY or has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.

Case 2    The result of MAXVAL(ARRAY, MASK = MASK) has a value equal to the maximum value of the elements of ARRAY corresponding to .TRUE. elements of MASK or has the value of the negative number of the largest magnitude supported by the processor for numbers of the same type and kind type parameter as ARRAY if there are no .TRUE. elements.

Case 3    If ARRAY has rank one, MAXVAL(ARRAY, DIM [,MASK]) has a value equal to that of MAXVAL(ARRAY [,MASK = MASK]).  Otherwise, the value of element $(s_1, s_2, ..., s_{\text{DIM}-1}, s_{\text{DIM}+1}, ..., s_n)$ of MAXVAL(ARRAY, DIM [,MASK]) is equal to the following:

```
MAXVAL(ARRAY(s₁, s₂, ..., s_DIM-1, :, s_DIM+1, ..., sₙ) [,
MASK = MASK(s₁, s₂, ..., s_DIM-1, :, s_DIM+1, ..., sₙ) ] )
```

**Examples**

Case 1          The value of `MAXVAL((/ 1, 2, 3 /))` is 3.

Case 2          `MAXVAL(C, MASK = C .LT. 0.0)` finds the maximum of the negative elements of `C`.

Case 3          If `B` is the array

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

then `MAXVAL(B, DIM = 1)` is [2, 4, 6] and `MAXVAL(B, DIM = 2)` is [5, 6].

# MCLOCK()

**Description**

Return time accounting for a program.

**Class**

Inquiry nonstandard function.

**Result Type**

Integer.

**Result Value**

The value returned, in units of microseconds, is the sum of the current process's user time and the user and system time of all its child processes.

# MERGE(TSOURCE, FSOURCE, MASK)

### Description

Choose alternative value according to the value of a mask.

### Class

Elemental function.

### Arguments

| | |
|---|---|
| TSOURCE | may be of any type. |
| FSOURCE | must be of the same type and type parameters as TSOURCE. |
| MASK | must be of type logical. |

### Result Type and Type Parameters

Same as TSOURCE.

### Result Value

The result is TSOURCE if MASK is .TRUE. and FSOURCE otherwise.

### Examples

If TSOURCE is the array $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}$

and FSOURCE is the array $\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}$

and MASK is the array $\begin{bmatrix} T & . & T \\ . & . & T \end{bmatrix}$

then where "T" represents .TRUE. and "." represents .FALSE., then
MERGE(TSOURCE, FSOURCE, MASK) is

$$\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$$

The value of MERGE(1.0, 0.0, K > 0) is 1.0 for K = 5 and 0.0 for
K = –2.

# MIN(A1, A2, A3, ...)

## Optional Arguments

A3, ...

## Description

Minimum value.

## Class

Elemental function.

## Arguments

The arguments must all be of the same type, which must be integer or real,
and they must all have the same kind type parameter.

## Result Type and Type Parameter

Same as the arguments.

## Result Value

The value of the result is that of the smallest argument.

### Examples

```
MIN(-9.0, 7.0, 2.0) has the value -9.0.
MIN(-0.4_HIGH, -1.0_HIGH/3) is -0.4_HIGH.
```

# MINEXPONENT(X)

### Description

Returns the minimum exponent in the model representing numbers of the same type and kind type parameter as the argument.

### Class

Inquiry function.

### Argument

X must be of type real.  It may be scalar or array valued.

### Result Type, Type Parameter, and Shape

Default integer scalar.

### Result Value

The result has the value $e_{min}$, as defined in the section "The Real Number System Model".

### Example

MINEXPONENT(X) has the value –125 for real X, whose model is described in "The Real Number System Model".

# MINLOC(ARRAY, MASK)

### Optional Argument

MASK

### Description

Returns the location of the first element of ARRAY having the minimum value of the elements identified by MASK.

### Class

Transformational function.

### Arguments

ARRAY               must be of type integer or real.  It must not be scalar.

MASK (optional)     must be of type logical and must be conformable with ARRAY.

### Result Type, Type Parameter, and Shape

The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

### Result Value

Case 1              If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the minimum value of all the elements of ARRAY.

                    The *i*th subscript returned lies in the range 1 to $e_i$, where $e_i$ is the extent of the *i*th dimension of ARRAY.

If more than one element has the minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, the value of the result is processor-dependent.

Case 2    If MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a .TRUE. element of MASK, whose value equals the minimum value of all such elements of ARRAY.

The *i*th subscript returned lies in the range 1 to $e_i$, where $e_i$ is the extent of the *i*th dimension of ARRAY. If more than one such element has the minimum value, the element whose subscripts are returned is the first such element taken in array element order.

If ARRAY has size zero or every element of MASK has the value .FALSE., the value of the result is processor-dependent.

In both cases, an element of the result is undefined if the processor cannot represent the value as a default integer.

### Examples

Case 1　　　　The value of MINLOC((/ 4, 3, 6, 3 /)) is [2].

　　　　　　　If the array B is declared

INTEGER, DIMENSION(4:7) :: B = (/ 8, 6, 3, 1 /)

　　　　　　　the value of MINLOC(B) is [4]

Case 2　　　　If A has the value

$$\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$$

then MINLOC(A, MASK = A .GT. -4) has the value [1, 4]. Note that this is true even if A has a declared lower bound other than 1.

# MINVAL(ARRAY, DIM, MASK)

### Optional Arguments

DIM, MASK

### Description

Minimum value of all the elements of ARRAY along dimension DIM corresponding to .TRUE. elements of MASK.

### Class

Transformational function.

## Arguments

ARRAY            must be of type integer or real. It must not be scalar.

DIM (optional)   must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where $n$ is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

MASK (optional)  must be of type logical and must be conformable with ARRAY.

## Result Type, Type Parameter, and Shape

The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or 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.

## Result Value

Case 1           The result of MINVAL(ARRAY) has a value equal to the minimum value of all the elements of ARRAY or has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.

Case 2           The result of MINVAL(ARRAY, MASK = MASK) has a value equal to the minimum value of the elements of ARRAY corresponding to .TRUE. elements of MASK or has the value of the positive number of the largest magnitude supported by the processor for numbers of the same type and kind type parameter as ARRAY if there are no .TRUE. elements.

Case 3      If `ARRAY` has rank one, `MINVAL(ARRAY, DIM [,MASK])` has a value equal to that of `MINVAL(ARRAY [,MASK = MASK])`. Otherwise, the value of element $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ of `MINVAL(ARRAY, DIM [,MASK])` is equal to the following:

`MINVAL(ARRAY(`$s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n$`) [, MASK= MASK(`$s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n$`) ] )`

### Examples

Case 1      The value of `MINVAL((/ 1, 2, 3 /))` is 1.

Case 2      `MINVAL(C, MASK = C .GT. 0.0)` forms the minimum of the positive elements of `C`.

Case 3      If B is the array

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

then `MINVAL(B, DIM = 1)` is [1, 3, 5] and `MINVAL(B, DIM = 2)` is [1, 2].

# MOD(A, P)

### Description

Remainder function.

### Class

Elemental function.

### Arguments

A            must be of type integer or real.

P            must be of the same type and kind type parameter as A.

### Result Type and Type Parameter

Same as A.

### Result Value

If P≠0, the value of the result is A – INT(A/P) * P. If P=0, the result is processor-dependent.

### Examples

MOD(3.0, 2.0) has the value 1.0.

MOD(8, 5) has the value 3.

MOD(-8, 5) has the value –3.

MOD(8, -5) has the value 3.

MOD(-8, -5) has the value –3.

MOD(2.0_HIGH, 3.0_HIGH) has the value 2.0_HIGH.

# MODULO(A, P)

### Description

Modulo function.

### Class

Elemental function.

## Arguments

A                         must be of type integer or real.

P                         must be of the same type and kind type parameter as A.

## Result Type and Type Parameter

Same as A.

## Result Value

Case 1                    A is of type integer.  If P≠0, MODULO(A, P) has the
                          value R such that A = Q × P + R, where Q is an
                          integer, the inequalities 0≤R<P hold if P>0, and P<R≤0
                          hold if P<0.  If P=0, the result is processor-dependent.

Case 2                    A is of type real.  If P≠0, the value of the result is
                          A −FLOOR (A / P) * P.  If P=0, the result is
                          processor-dependent.

## Examples

MODULO(8, 5) has the value 3.

MODULO(-8, 5) has the value 2.

MODULO(8, -5) has the value –2.

MODULO(-8, -5) has the value –3.

MODULO(3.0, 2.0) has the value 1.0.

MODULO(2.0_HIGH, 3.0_HIGH) has the value 2.0_HIGH.

# MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

### Description

Copies a sequence of bits from one data object to another.

### Class

Elemental subroutine.

### Arguments

FROM
: must be of type integer.  It is an `INTENT(IN)` argument.

FROMPOS
: must be of type integer and nonnegative. It is an `INTENT(IN)` argument. `FROMPOS + LEN` must be less than or equal to `BIT_SIZE(FROM)`. The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

LEN
: must be of type integer and nonnegative. It is an `INTENT(IN)` argument.

TO
: must be a variable of type integer with the same kind type parameter value as `FROM` and may be the same variable as `FROM`.  It is an `INTENT(INOUT)` argument.

  `TO` is set by copying the sequence of bits of length `LEN`, starting at position `FROMPOS` of `FROM` to position `TOPOS` of `TO`.  No other bits of `TO` are altered.  On return, the `LEN` bits of `TO` starting at `TOPOS` are equal to the value that the `LEN` bits of `FROM` starting at `FROMPOS` had on entry.

  The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

TOPOS             must be of type integer and nonnegative. It is an
                  `INTENT(IN)` argument. `TOPOS` + `LEN` must be less than
                  or equal to `BIT_SIZE(TO)`.

### Examples

If `TO` has the initial value 6, the value of `TO` after the statement `CALL
MVBITS(7, 2, 2, TO, 0)` is 5.

After the statement

```
CALL MVBITS (PATTERN, 0_SHORT, 1_SHORT,
  PATTERN, 7_SHORT)
```

is executed, the integer variable `PATTERN` of kind `SHORT` has a leading bit
that is identical to its terminal bit.

# NEAREST(X, S)

### Description

Returns the nearest different machine representable number in a given
direction.

### Class

Elemental function.

### Arguments

X             must be of type real.

S             must be of type real and not equal to zero.

### Result Type and Type Parameter

Same as `X`.

**Result Value**

The result has a value equal to the machine representable number distinct from X and nearest to it in the direction of the infinity with the same sign as S.

**Example**

NEAREST(3.0, 2.0) has the value $3+2^{-22}$ on a machine whose representation is that of the model described in the section "The Real Number System Model".

# NINT(A, KIND)

**Optional Argument**

KIND

**Description**

Nearest integer.

**Class**

Elemental function.

**Arguments**

A                          must be of type real.

KIND (optional)      must be a scalar integer initialization expression.

**Result Type and Type Parameter**

Integer.  If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default integer type.

**Result Value**

If A>0, NINT(A) has the value INT(A+0.5); if A≤0, NINT(A) has the value INT(A−0.5). The result is undefined if the processor cannot represent the result in the specified integer type.

**Examples**

NINT(2.783) has the value 3.

NINT(-1.99999999999_HIGH) has the value -2.

# NOT(I)

**Description**

Performs a bitwise logical complement.

**Class**

Elemental function.

**Argument**

I must be of type integer.

**Result Type and Type Parameter**

Same as I.

### Result Value

The result has the value obtained by complementing `I` bit-by-bit according to the following truth table:

| I | NOT(I) |
|---|--------|
| 1 | 0      |
| 0 | 1      |

The model for the interpretation of an integer value as a sequence of bits is in the section "The Bit Model".

### Example

If `I` is an integer of kind `SHORT` and has a value that is equal to 01010101 (base 2), `NOT(I)` has a value that is equal to 10101010 (base 2).

# OR(I, J)

## Description

Bitwise logical OR.

## Class

Elemental nonstandard function.

## Arguments.

| | |
|---|---|
| `I` | must be of type integer. |
| `J` | must be of type integer with the same kind type parameter as `I`. |

### Result Type and Type Parameter

Same as I.

### Result Value

The result has the value obtained by performing an OR on I and J bit-by-bit according to the following truth table:

| I | J | OR(I, J) |
|---|---|----------|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

The model for interpreting an integer value as a sequence of bits is in the section "The Bit Model".

### Example

OR(3, 5) is 7. (Binary 0011 OR with binary 0101 is binary 0111.)

# PACK(ARRAY, MASK, VECTOR)

### Optional Argument

VECTOR

### Description

Pack an array into an array of rank one under the control of a mask.

### Class

Transformational function.

### Arguments

ARRAY           may be of any type.  It must not be scalar.

MASK           must be of type logical and must be conformable with ARRAY.

VECTOR (optional) must be of the same type and type parameters as ARRAY and must have rank one.  VECTOR must have at least as many elements as there are .TRUE. elements in MASK. If MASK is scalar with the value .TRUE., VECTOR must have at least as many elements as there are in ARRAY.

### Result Type, Type Parameter, and Shape

The result is an array of rank one with the same type and type parameters as ARRAY.  If VECTOR is present, the result size is that of VECTOR; otherwise, the result size is the number $t$ of .TRUE. elements in MASK unless MASK is scalar with the value .TRUE., in which case the result size is the size of ARRAY.

### Result Value

Element $i$ of the result is the element of ARRAY that corresponds to the $i$th .TRUE. element of MASK, taking elements in array element order, for $i = 1$, 2, ..., $t$. If VECTOR is present and has size $n>t$, element $i$ of the result has the value VECTOR ($i$), for $i = t+1, \ldots, n$.

### Examples

The nonzero elements of an array M with the value

$$\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

may be "gathered" by the function PACK.

The result of PACK(M, MASK = M .NE. 0) is [9, 7].

The result of PACK(M, M .NE. 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is [9, 7, 6, 8, 10, 12].

## PRECISION(X)

### Description

Returns the decimal precision in the model representing real numbers with the same kind type parameter as the argument.

### Class

Inquiry function.

### Argument

X must be of type real or complex. It may be scalar or array valued.

### Result Type, Type Parameter, and Shape

Default integer scalar.

### Result Value

The result has the value `INT((p-1) * LOG10(b))+k`. The values of $b$ and $p$ are as defined in the section "The Real Number System Model" for the model representing real numbers with the same kind type parameter as X. The value of $k$ is 1 if $b$ is an integral power of 10 and 0 otherwise.

**Example.** `PRECISION(X)` has the value `INT(23 * LOG10(2.)) = INT(6.92...) = 6` for real X whose model is described in the section "The Real Number System Model".

# PRESENT(A)

### Description

Determine whether an optional argument is present.

### Class

Inquiry function.

### Argument

A must be the name of an optional dummy argument that is accessible in the procedure in which the PRESENT function reference appears.

### Result Type and Type Parameters

Default logical scalar.

### Result Value

The result has the value .TRUE. if A is present and otherwise has the value .FALSE..

### Example

```
SUBROUTINE SUB (A, B, EXTRA)
   REAL A, B, C
   REAL, OPTIONAL :: EXTRA
   . . .
   IF (PRESENT (EXTRA)) THEN
      C = EXTRA
   ELSE
      C = (A+B)/2
   END IF
   . . .
END
```

If SUB is called with the statement

CALL SUB (10.0, 20.0, 30.0)

then C is set to 30.0. If SUB is called with the statement

CALL SUB (10.0, 20.0)

then C is set to 15.0.

An optional argument that is not present must not be referenced or defined or supplied as a nonoptional actual argument, except as the argument of the PRESENT intrinsic function.

# PRODUCT(ARRAY, DIM, MASK)

### Optional Arguments

DIM, MASK

### Description

Product of all the elements of ARRAY along dimension DIM corresponding to the .TRUE. elements of MASK.

### Class

Transformational function.

## Arguments

ARRAY            must be of type integer, real, or complex.  It must not be scalar.

DIM (optional)   must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where $n$ is the rank of ARRAY.  The corresponding actual argument must not be an optional dummy argument.

MASK (optional)  must be of type logical and must be conformable with ARRAY.

## Result Type, Type Parameter, and Shape

The result is of the same type and kind type parameter as ARRAY.  It is scalar if DIM is absent or 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.

## Result Value

Case 1           The result of PRODUCT(ARRAY) has a value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has the value one if ARRAY has size zero.

Case 2           The result of PRODUCT(ARRAY, MASK = msk) has a value equal to a processor-dependent approximation to the product of the elements of ARRAY corresponding to the .TRUE. elements of msk or has the value one if there are no .TRUE. elements.

Case 3           If ARRAY has rank one, PRODUCT(ARRAY, DIM [,msk]) has a value equal to that of PRODUCT(ARRAY [,MASK = ask]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of PRODUCT(ARRAY, DIM [,msk]) is equal to the following:

                 PRODUCT(ARRAY($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$) [, MASK = msk($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$) ] )

**Examples**

| | |
|---|---|
| Case 1 | The value of `PRODUCT((/ 1, 2, 3 /))` and `PRODUCT((/ 1, 2, 3 /), DIM=1)` is 6. |
| Case 2 | `PRODUCT(C, MASK = C .GT. 0.0)` forms the product of the positive elements of `C`. |
| Case 3 | If `B` is the array |

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

then `PRODUCT(B, DIM = 1)` is [2, 12, 30] and `PRODUCT(B, DIM = 2)` is [15, 48].

# RADIX(X)

**Description**

Returns the base of the model representing numbers of the same type and kind type parameter as the argument.

**Class**

Inquiry function.

**Argument**

`X` must be of type integer or real. It may be scalar or array valued.

**Result Type, Type Parameter, and Shape**

Default integer scalar.

### Result Value

The result has the value *r* if X is of type integer and the value *b* if X is of type real, where *r* and *b* are as defined in the section "The Real Number System Model".

### Example

RADIX(X) has the value 2 for real X whose model is described in the section "The Real Number System Model".

# RANDOM_NUMBER(HARVEST)

### Description

Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 \leq x < 1$.

### Class

Subroutine.

### Argument

HARVEST must be of type real. It is an INTENT(OUT) argument. It may be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution in the interval $0 \leq x < 1$.

### Examples

```
REAL X, Y (10, 10)
! Initialize X with a pseudorandom number
CALL RANDOM_NUMBER (HARVEST = X)
CALL RANDOM_NUMBER (Y)
! X & Y contain
! uniformly distributed random numbers
```

# RANDOM_SEED(SIZE, PUT, GET)

### Optional Arguments

SIZE, PUT, GET

### Description

Restarts or queries the pseudorandom number generator used by
RANDOM_NUMBER.

### Class

Subroutine.

### Arguments

There must either be exactly one or no arguments present.

SIZE (optional)      must be scalar and of type default integer. It is an
                     INTENT(OUT) argument. It is set to the number $N$ of
                     integers that the processor uses to hold the value of the
                     seed.

PUT (optional)       must be a default integer array of rank one and size $\geq N$.
                     It is an INTENT(IN) argument. It is used by the
                     processor to set the seed value.

GET (optional)     must be a default integer array of rank one and size $\geq N$.
It is an INTENT(OUT) argument. It is set by the
processor to the current value of the seed. If no
argument is present, the processor sets the seed to a
processor-dependent value.

## Examples

```
CALL RANDOM_SEED
! Processor initialization
CALL RANDOM_SEED (SIZE = K)            ! Sets K = N
CALL RANDOM_SEED (PUT = SEED (1 : K)) ! Set user seed
CALL RANDOM_SEED (GET = OLD (1 : K))  ! Read current
seed
```

# RANGE(X)

### Description

Returns the decimal exponent range in the model representing integer or
real numbers with the same kind type parameter as the argument.

### Class

Inquiry function.

### Argument

X must be of type integer, real, or complex. It may be scalar or array valued.

### Result Type, Type Parameter, and Shape

Default integer scalar.

**Result Value**

Case 1    For an integer argument, the result has the value INT(LOG10(*huge*)), where *huge* is the largest positive integer in the model representing integer numbers with same kind type parameter as X (see the section "The Integer Number System Model").

Case 2    For a real or complex argument, the result has the value INT(MIN(LOG10(*huge*), -LOG10(*tiny*))), where *huge* and *tiny* are the largest and smallest positive numbers in the model representing real numbers with the same value for the kind type parameter as X (see the section "The Real Number System Model").

**Example**

RANGE(X) has the value 38 for real X, whose model is described in "The Real Number System Model", because in this case $huge = (1 - 2^{-24}) \times 2^{127}$ and $tiny = 2^{-127}$.

# REAL(A, KIND)

**Optional Argument**

KIND

**Description**

Convert to real type.

**Class**

Elemental function.

## Arguments

A                         must be of type integer, real, or complex.

KIND (optional)    must be a scalar integer initialization expression.

## Result Type and Type Parameter

### Real

Case 1            If A is of type integer or real and KIND is present, the
                  kind type parameter is that specified by KIND.

                  If A is of type integer or real and KIND is not present, the
                  kind type parameter is the processor-dependent kind
                  type parameter for the default real type.

Case 2            If A is of type complex and KIND is present, the kind
                  type parameter is that specified by KIND.

                  If A is of type complex and KIND is not present, the kind
                  type parameter is the kind type parameter of A.

### Result Value

Case 1            If A is of type integer or real, the result is equal to a
                  processor-dependent approximation to A.

Case 2            If A is of type complex, the result is equal to a
                  processor-dependent approximation to the real part of A.

## Examples

REAL(-3) has the value –3.0.

REAL(Z) has the same kind type parameter and the same value as the real
part of the complex variable Z.

REAL(2.0_HIGH/3.0) is 0.66666666666666 with kind HIGH.

# REPEAT(STRING, NCOPIES)

### Description

Concatenate several copies of a string.

### Class

Transformational function.

### Arguments

STRING          must be scalar and of type character.

NCOPIES         must be scalar and of type integer. Its value must not be
                negative.

### Result Type, Type Parameter, and Shape

Character scalar of length NCOPIES times that of STRING, with the same
kind type parameter as STRING.

### Result Value

The value of the result is the concatenation of NCOPIES copies of STRING.

### Examples

REPEAT('H', 2) has the value HH.

REPEAT('XYZ', 0) has the value of a zero-length string.

# RESHAPE(SOURCE, SHAPE, PAD, ORDER)

### Optional Arguments

PAD, ORDER

### Description

Constructs an array of a specified shape from the elements of a given array.

### Class

Transformational function.

### Arguments

SOURCE            may be of any type. It must be array valued. If PAD is
                  absent or of size zero, the size of SOURCE must be
                  greater than or equal to PRODUCT(SHAPE). The size of
                  the result is the product of the values of the elements of
                  SHAPE.

SHAPE             must be of type integer, rank one, and constant size. Its
                  size must be positive and less than 8. It must not have
                  an element whose value is negative.

PAD (optional)    must be of the same type and type parameters as
                  SOURCE. PAD must be array valued.

ORDER (optional)  must be of type integer, must have the same shape as
                  SHAPE, and its value must be a permutation of [1, 2, ...,
                  $n$], where $n$ is the size of SHAPE. If absent, it is as if it
                  were present with value [1, 2, ..., $n$].

### Result Type, Type Parameter, and Shape

The result is an array of shape SHAPE (that is, SHAPE(RESHAPE(SOURCE,
SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and
type parameters as SOURCE.

**Result Value**

The elements of the result, taken in permuted subscript order ORDER(1), ..., ORDER(*n*), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

**Examples**

```
RESHAPE((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /)) has the
value
```

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

```
RESHAPE((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 4 /), (/ 0, 0 /),
(/ 2, 1 /)) has the value
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$$

# RNUM(I)

**Description**

Convert character to real type.

**Class**

Elemental nonstandard function.

**Argument**

I must be of type character.

**Result**

Default real type.

**Example**

`RNUM("821.003")` is 821.003 of type default real.

# RRSPACING(X)

### Description

Returns the reciprocal of the relative spacing of model numbers near the argument value.

### Class

Elemental function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has the value $|X \times b^{-e}| \times b^{p}$, where $b$, $e$, and $p$ are as defined in the section "The Real Number System Model".

### Example

`RRSPACING(-3.0)` has the value $0.75 \times 2^{24}$ for reals whose model is described in the section "The Real Number System Model".

# RSHFT(I, SHIFT)

### Description

Bitwise right shift.

### Class

Elemental nonstandard function.

For details see "RSHFT" on page 120.

# RSHIFT(I, SHIFT)

### Description

Bitwise right shift.

### Class

Elemental nonstandard function.

For details see "RSHIFT" on page 121.

# SCALE(X, I)

### Description

Returns $X \times b^I$ where $b$ is the base in the model representation of X (see the section "The Real Number System Model").

### Class

Elemental function.

### Arguments

X                   must be of type real.

I                   must be of type integer.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has the value $X \times b^I$, where $b$ is defined in the section "The Real Number System Model", provided this result is within range; if not, the result is processor dependent.

### Example

SCALE(3.0, 2) has the value 12.0 for reals whose model is described in "The Real Number System Model".

# SCAN(STRING, SET, BACK)

**Optional Argument**

BACK

**Description**

Scan a string for any one of the characters in a set of characters.

**Class**

Elemental function.

**Arguments**

| | |
|---|---|
| STRING | must be of type character. |
| SET | must be of type character with the same kind type parameter as STRING. |
| BACK (optional) | must be of type logical. |

**Result Type and Type Parameter**

Default integer.

**Result Value**

| | |
|---|---|
| Case 1 | If BACK is absent or is present with the value .FALSE. and if STRING contains at least one character that is in SET, the value of the result is the position of the leftmost character of STRING that is in SET. |
| Case 2 | If BACK is present with the value .TRUE. and if STRING contains at least one character that is in SET, the value of the result is the position of the rightmost character of STRING that is in SET. |

Case 3          The value of the result is zero if no character of STRING
                is in SET or if the length of STRING or SET is zero.

**Examples**

Case 1          SCAN('FORTRAN', 'TR') has the value 3.

Case 2          SCAN('FORTRAN', 'TR', BACK = .TRUE.) has the
                value 5.

Case 3          SCAN('FORTRAN', 'BCD') has the value 0.

# SELECTED_INT_KIND(R)

### Description

Returns a value of the kind type parameter of an integer data type that
represents all integer values $n$ with $-10^R < n < 10^R$.

### Class

Transformational function.

### Argument

R must be scalar and of type integer.

### Result Type, Type Parameter, and Shape

Default integer scalar.

### Result Value

The result has a value equal to the value of the kind type parameter of an
integer data type that represents all values $n$ in the range of values $n$ with
$-10^R < n < 10^R$, or if no such kind type parameter is available on the
processor, the result is –1.

If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range, unless there are several such values, in which case the smallest of these kind values is returned.

### Example

SELECTED_INT_KIND(6) has the value KIND(0) on a machine that supports a default integer representation method with $r=2$ and $q=31$ as defined in the model for the integer number systems in "The Integer Number System Model".

# SELECTED_REAL_KIND(P, R)

### Optional Arguments

P, R

### Description

Returns a value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range of at least R.

### Class

Transformational function.

### Arguments

At least one argument must be present.

P (optional)          must be scalar and of type integer.

R (optional)          must be scalar and of type integer.

### Result Type, Type Parameter, and Shape

Default integer scalar.

**Result Value**

The result has a value equal to a value of the kind type parameter of a real data type with decimal precision, as returned by the function PRECISION, of at least P digits and a decimal exponent range, as returned by the function RANGE, of at least R.

If no such kind type parameter is available on the processor, the result is –1 if the precision is not available, –2 if the exponent range is not available, and –3 if neither is available.

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

**Example**

SELECTED_REAL_KIND(6, 70) has the value KIND(0.0) on a machine that supports a default real approximation method with $p=16$, $p=6$, $e_{min}=-64$, and $e_{max}=63$ as defined in the model for the real number system in the section "The Real Number System Model".

# SET_EXPONENT(X, I)

**Description**

Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

**Class**

Elemental function.

### Arguments

X          must be of type real.

I          must be of type integer.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has the value $X \times b^{I-e}$, where $b$ and $e$ are as defined in the section "The Real Number System Model", provided this result is within range; if not, the result is processor-dependent.

If X has value zero, the result has value zero.

### Example

SET_EXPONENT(3.0, 1) has the value 1.5 for reals whose model is as described in the section "The Real Number System Model".

# SHAPE(SOURCE)

### Description

Returns the shape of an array or a scalar.

### Class

Inquiry function.

### Argument

SOURCE may be of any type. It may be array valued or scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It must not be an assumed-size array.

### Result Type, Type Parameter, and Shape

The result is a default integer array of rank one whose size is equal to the rank of SOURCE.

### Result Value

The value of the result is the shape of SOURCE.

### Examples

The value of SHAPE(A(2:5, −1:1) ) is [4, 3].

The value of SHAPE(3) is the rank-one array of size zero.

# SIGN(A, B)

### Description

Absolute value of A times the sign of B.

### Class

Elemental function.

### Arguments

| | |
|---|---|
| *A* | must be of type integer or real. |
| *B* | must be of the same type and kind type parameter as A. |

### Result Type and Type Parameter

Same as A.

### Result Value

The value of the result is |A| if B $\geq$ 0 and –|A| if B < 0.

### Example

SIGN(–3.0, 2.0) has the value 3.0.

# SIN(X)

### Description

Sine function in radians.

### Class

Elemental function.

### Argument

X must be of type real or complex.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to sin(X).

If X is of type real, it is regarded as a value in radians.

If X is of type complex, its real part is regarded as a value in radians.

### Examples

SIN(1.0) has the value 0.84147098.

SIN((0.5_HIGH, 0.5)) has the value $0.54061268571316 + 0.45730415318425i$ with kind HIGH.

# SIND(X)

### Description

Sine function that accepts input in degrees.

### Class

Elemental nonstandard function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to sin(X).

### Examples

SIND(0.0) has the value 0.0.

SIND(30.0) has the value 0.5.

# SINH(X)

### Description

Hyperbolic sine function.

### Class

Elemental function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

The result has a value equal to a processor-dependent approximation to sinh(X).

### Examples

SINH(1.0) has the value 1.1752012.

SINH(0.5_HIGH) has the value 0.52109530549375 with kind HIGH.

# SIZE(ARRAY, DIM)

### Optional Argument

DIM

### Description

Returns the extent of an array along a specified dimension or the total number of elements in the array.

### Class

Inquiry function.

### Arguments.

ARRAY          may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. If ARRAY is an assumed-size array, DIM must be present with a value less than the rank of ARRAY.

DIM (optional)  must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where $n$ is the rank of ARRAY.

### Result Type, Type Parameter, and Shape

Default integer scalar.

### Result Value

The result has a value equal to the extent of dimension DIM of ARRAY or, if DIM is absent, the total number of elements of ARRAY.

### Examples

The value of SIZE(A(2:5, -1:1), DIM=2) is 3.

The value of SIZE(A(2:5, -1:1) ) is 12.

# SPACING(X)

### Description

Returns the absolute spacing of model numbers near the argument value.

### Class

Elemental function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

### Result Value

If X is not zero, the result has the value $b^{e-p}$, where $b$, $e$, and $p$ are as defined in the section "The Real Number System Model", provided this result is within range; otherwise, the result is the same as that of TINY(X).

### Example

SPACING(3.0) has the value $2^{-22}$ for reals whose model is described in the section "The Real Number System Model".

# SPREAD(SOURCE, DIM, NCOPIES)

### Description

Replicates an array by adding a dimension. Broadcasts several copies of SOURCE along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater.

### Class

Transformational function.

### Arguments

SOURCE          may be of any type. It may be scalar or array valued. The rank of SOURCE must be less than 7.

DIM             must be scalar and of type integer with value in the range $1 \leq \text{DIM} \leq n + 1$, where $n$ is the rank of SOURCE.

NCOPIES         must be scalar and of type integer.

### Result Type, Type Parameter, and Shape

The result is an array of the same type and type parameters as SOURCE and of rank $n + 1$, where $n$ is the rank of SOURCE.

Case 1          If SOURCE is scalar, the shape of the result is (MAX(NCOPIES, 0)).

Case 2          If SOURCE is array valued with shape $(d_1, d_2, \ldots, d_n)$, the shape of the result is $(d_1, d_2, \ldots, d_{\text{DIM}-1},$ MAX(NCOPIES, 0), $d_{\text{DIM}}, \ldots, d_n)$.

**Result Value**

Case 1          If SOURCE is scalar, each element of the result has a value equal to SOURCE.

Case 2          If SOURCE is array valued, the element of the result with subscripts $(r_1, r_2, \ldots, r_{n+1})$ has the value SOURCE$(r_1, r_2, \ldots, r_{DIM-1}, r_{DIM+1}, \ldots, r_{n+1})$.

**Examples**

Case 1          SPREAD("A", 1, 3) is the character array (/ "A", "A", "A" /).

Case 2          If A is the array [2, 3, 4], SPREAD(A, DIM=1, NCOPIES=NC) is the array

$$
\begin{bmatrix}
2 & 3 & 4 \\
2 & 3 & 4 \\
2 & 3 & 4
\end{bmatrix}
$$

if NC has the value 3 and is a zero-sized array if NC has the value 0.

# SQRT(X)

**Description**

Square root.

**Class**

Elemental function.

**Argument**

X must be of type real or complex. If X is real, its value must be greater than or equal to zero.

**Result Type and Type Parameter**

Same as X.

**Result Value**

The result has a value equal to a processor-dependent approximation to the square root of X.

A result of type complex 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.

**Examples**

SQRT(4.0) has the value 2.0.

SQRT(5.0_HIGH) has the value 2.23606774998 with kind HIGH.

# SUM(ARRAY, DIM, MASK)

**Optional Arguments**

DIM, MASK

**Description**

Sum all the elements of ARRAY along dimension DIM corresponding to the .TRUE. elements of MASK.

**Class**

Transformational function.

### Arguments

| | |
|---|---|
| ARRAY | must be of type integer, real, or complex. It must not be scalar. |
| DIM (optional) | must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where $n$ is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument. |
| MASK (optional) | must be of type logical and must be conformable with ARRAY. |

### Result Type, Type Parameter, and Shape

The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent of 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.

### Result Value

| | |
|---|---|
| Case 1 | The result of SUM(ARRAY) has a value equal to a processor-dependent approximation to the sum of all the elements of ARRAY or has the value zero if ARRAY has size zero. |
| Case 2 | The result of SUM(ARRAY, MASK = msk) has a value equal to a processor-dependent approximation to the sum of the elements of ARRAY corresponding to the .TRUE. elements of msk or has the value zero if there are no .TRUE. elements. |
| Case 3 | If ARRAY has rank one, SUM(ARRAY, DIM [,msk]) has a value equal to that of SUM(ARRAY [,MASK = msk]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of SUM(ARRAY, DIM [,msk]) is equal to the following: |

SUM(ARRAY$(s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n)$ [, MASK=msk$(s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n)$])

**Examples**

Case 1        The value of SUM((/ 1, 2, 3 /)) and SUM((/ 1, 2, 3 /), DIM=1) is 6.

Case 2        SUM(C, MASK= C .GT. 0.0) forms the arithmetic sum of the positive elements of C.

Case 3        If B is the array

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

then SUM(B, DIM = 1) is [3, 7, 11] and SUM(B, DIM = 2) is [9, 12].

# SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)

**Optional Arguments**

COUNT, COUNT_RATE, COUNT_MAX

**Description**

Returns integer data from a real-time clock.

**Class**

Subroutine.

**Arguments**

COUNT (optional)        must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to a processor-dependent value based on the current

value of the processor clock or to −HUGE(0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.

COUNT_RATE (optional)  must be scalar and 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 (optional)  must be scalar and of type default integer.  It is an INTENT(OUT) argument.  It is set to the maximum value that COUNT can have, or to zero if there is no clock.

### Example

If the processor clock is a 24-hour clock that registers time in 1-second intervals, at 11:30 A.M. the reference

```
CALL SYSTEM_CLOCK (COUNT = C, COUNT_RATE = R, COUNT_M
AX = M)
```

sets C = 11 ¥ 3600 + 30 ¥ 60 = 41400, R = 1, and M = 24 ¥ 3600 − 1 = 86399.

# TAN(X)

### Description

Tangent function, which accepts the input in radians.

### Class

Elemental function.

**Argument**

X must be of type real.

**Result Type and Type Parameter**

Same as X.

**Result Value**

The result has a value equal to a processor-dependent approximation to tan(X), with X regarded as a value in radians.

**Examples**

TAN(1.0) has the value 1.5574077.

TAN(2.0_HIGH) has the value –2.1850398632615 with kind HIGH.

# TAND(X)

### Description

Tangent function that accepts the input in degrees.

### Class

Elemental nonstandard function.

### Argument

X must be of type real.

### Result Type and Type Parameter

Same as X.

**Result Value**

The result has a value equal to a processor-dependent approximation to tan(X).

**Examples**

TAND(0.0) has the value 0.0.

TAND(45.0) has the value 1.0.

TAND(135.0) has the value -1.0.

# TANH(X)

**Description**

Hyperbolic tangent function.

**Class**

Elemental function.

**Argument**

X must be of type real.

**Result Type and Type Parameter**

Same as X.

**Result Value**

The result has a value equal to a processor-dependent approximation to tanh(X).

**Examples**

TANH(1.0) has the value 0.76159416.

`TANH(2.0_HIGH)` has the value 0.96402758007582 with kind `HIGH`.

# TINY(X)

### Description

Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

### Class

Inquiry function.

### Argument

*X* must be of type real.  It may be scalar or array valued.

### Result Type, Type Parameter, and Shape

Scalar with the same type and kind type parameter as `X`.

### Result Value

The result has the value

$b^{e_{min}-1}$

where $b$ and $e_{min}$ are as defined in the section "The Real Number System Model".

### Example

`TINY(X)` has the value $2^{-127}$ for real `X`, whose model is described in the section "The Real Number System Model".

# TRANSFER(SOURCE, MOLD, SIZE)

## Optional Argument

SIZE

## Description

Returns a result with a physical representation identical to that of SOURCE but interpreted with the type and type parameters of MOLD.

## Class

Transformational function.

## Arguments

SOURCE            may be of any type and may be scalar or array valued.

MOLD              may be of any type and may be scalar or array valued.

SIZE  (optional)   must be scalar and of type integer.  The corresponding actual argument must not be an optional dummy argument.

## Result Type, Type Parameter, and Shape

The result is of the same type and type parameters as MOLD.

Case 1            If MOLD is a scalar and SIZE is absent, the result is a scalar.

Case 2            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 its physical representation is not shorter than that of SOURCE.

Case 3            If SIZE is present, the result is array valued of rank one and size SIZE.

**Result Value**

If the physical representation of the result has the same length as that of SOURCE, the physical representation of the result is that of SOURCE.

If the physical representation of the result is longer than that of SOURCE, the physical representation of the leading part is that of SOURCE and the remainder is 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. If D and E are scalar variables such that the physical representation of D is as long as or longer than that of E, the value of TRANSFER(TRANSFER(E, D), E) must be the value of E.

If D is an array and E is an array of rank one, the value of TRANSFER(TRANSFER(E, D), E, SIZE(E)) must be the value of E.

**Examples**

Case 1    TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000.

Case 2    TRANSFER((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /)) is a complex rank-one array of length two whose first element is (1.1, 2.2) and whose second element has a real part with the value 3.3. The imaginary part of the second element is undefined.

Case 3    TRANSFER((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /), 1) has the value $1.1 + 2.2i$, which is a rank-one array with one complex element.

# TRANSPOSE(MATRIX)

### Description

Transpose an array of rank two.

### Class

Transformational function.

### Argument

MATRIX may be of any type and must have rank two.

Result Type,Type Parameters, and Shape. The result is an array of the same type and type parameters as MATRIX and with rank two and shape *(n, m)* where *(m, n)* is the shape of MATRIX.

### Result Value

Element *(i, j)* of the result has the value MATRIX(*j, i*), *i*= 1, 2, ..., *n*; *j*= 1, 2, ..., *m*.

### Example

If A is the array

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

then TRANSPOSE(A) has the value

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

# TRIM(STRING)

### Description

Returns the argument with trailing blank characters removed.

### Class

Transformational function.

### Argument

STRING must be of type character and must be a scalar.

### Result Type and Type Parameters

Character with the same kind type parameter value as STRING and with a
length that is the length of STRING less the number of trailing blanks in
STRING.

### Result Value

The value of the result is the same as STRING except any trailing blanks are
removed. If STRING contains no nonblank characters, the result has zero
length.

### Example

TRIM('ƀAƀBƀƀ') is 'ƀAƀB'.

# UBOUND(ARRAY, DIM)

**Optional Argument**

DIM

**Description**

Returns all the upper bounds of an array or a specified upper bound.

**Class**

Inquiry function.

**Arguments**

ARRAY              may be of any type.  It must not be scalar.  It must not be
                   a pointer that is disassociated or an allocatable array that
                   is not allocated.  If ARRAY is an assumed-size array, DIM
                   must be present with a value less than the rank of ARRAY.

DIM (optional)     must be scalar and of type integer with a value in the
                   range $1 \le \text{DIM} \le n$, where $n$ is the rank of ARRAY.  The
                   corresponding actual argument must not be an optional
                   dummy argument.

**Result Type, Type Parameter, and Shape.**

The result is of type default integer.  It is scalar if DIM is present; otherwise,
the result is an array of rank one and size $n$, where $n$ is the rank of ARRAY.

**Result Value**

Case 1
For an array section or for an array expression, other than a whole array or array structure component, UBOUND(ARRAY, DIM) has a value equal to the number of elements in the given dimension; otherwise, it has a value equal to the upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero.

Case 2
UBOUND(ARRAY) has a value whose *i*th component is equal to UBOUND(ARRAY, *i*), for *i* = 1, 2, ..., *n*, where *n* is the rank of ARRAY.

**Examples**

If the following statements are processed

```
REAL, TARGET :: A (2:3, 7:10)
REAL, POINTER, DIMENSION (:,:) :: B, C, D
B => A;  C => A(:,:)
ALLOCATE (D(-3:3,-7:7))
```

then

- UBOUND(A) is [3, 10]
- UBOUND(A, DIM = 2) is 10
- UBOUND(B) is [3, 10]
- UBOUND(C) is [2, 4]
- UBOUND(D) is [3, 7]

# UNPACK(VECTOR, MASK, FIELD)

### Description

Unpack an array of rank one into an array under the control of a mask.

### Class

Transformational function.

### Arguments

| | |
|---|---|
| VECTOR | may be of any type. It must have rank one. Its size must be at least *t* where *t* is the number of .TRUE. elements in MASK. |
| MASK | must be array valued and of type logical. |
| FIELD | must be of the same type and type parameters as VECTOR and must be conformable with MASK. |

### Result Type, Type Parameter, and Shape

The result is an array of the same type and type parameters as VECTOR and the same shape as MASK.

### Result Value

The element of the result that corresponds to the *i*th .TRUE. 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 a value equal to FIELD if FIELD is scalar or to the corresponding element of FIELD if it is an array.

## Examples

Specific values may be "scattered" to specific positions in an array by using
UNPACK. If M is the array

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then V is the array [1, 2, 3],

and Q is the logical mask

$$\begin{bmatrix} . & T & . \\ T & . & . \\ . & . & T \end{bmatrix}$$

where "T" represents .TRUE. and "." represents .FALSE., then the result
of UNPACK(V, MASK = Q, FIELD = M) has the value

$$\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

and the result of UNPACK(V, MASK = Q, FIELD = 0) has the value

$$\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

# VERIFY(STRING, SET, BACK)

### Optional Argument

`BACK`

### Description

Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

### Class

Elemental function.

### Arguments

| | |
|---|---|
| `STRING` | must be of type character. |
| `SET` | must be of type character with the same kind type parameter as `STRING`. |
| `BACK` (optional) | must be of type logical. |

### Result Type and Type Parameter

Default integer.

**Result Value**

Case 1    If BACK is absent or present with the value .FALSE. and if STRING contains at least one character that is not in SET, the value of the result is the position of the leftmost character of STRING that is not in SET.

Case 2    If BACK is present with the value .TRUE. and if STRING contains at least one character that is not in SET, the value of the result is the position of the rightmost character of STRING that is not in SET.

Case 3    The value of the result is zero if each character in STRING is in SET or if STRING has zero length.

**Examples**

Case 1    VERIFY('ABBA', 'A') has the value 2.

Case 2    VERIFY('ABBA', 'A', BACK = .TRUE.) has the value 3.

Case 3    VERIFY('ABBA', 'AB') has the value 0.

# XOR(I, J)

### Description

Bitwise exclusive OR.

### Class

Elemental nonstandard function.

### Arguments

| | |
|---|---|
| `I` | must be of type integer. |
| `J` | must be of type integer with the same kind type parameter as `I`. |

### Result Type and Type Parameter

Same as `I`.

**NOTE.** *See the section "*IXOR(I, J)*" for result value information and examples.*

# *Portability Functions* 2

This chapter describes the functions (in alphabetical order) that comprise the portability library (`libPEPCF90.lib`). These functions are made available to the compiler when you invoke the `/4Yportlib` option. The function prototypes are described using the INTERFACE block, which provide the required information to complete a call to the specified function. For descriptions of the INTERFACE block and the `/4Yportlib` option, see the *Intel® Fortran Compiler User's Guide*

When using these functions, take the following considerations in mind:

- These functions are not a built-in part of the Fortran language. If you have `FUNCTION`, `SUBROUTINE`, or `COMMON` block names in your program that conflict with a particular portability function that you wish to use, you may need to change the name of the global entity in your program, in order to access the portability function.

- These functions are in user name space, and are linked only when you use the `/4Yportlib` switch, and only then when there is no global definition in your program that satisfies the global reference.

- You do not have to insert a `USE` statement to interface to these functions, as with some other vendor's products. You should use an `INTERFACE` block in your program to describe the interface to the routine in the portability library that you are calling. Without an `INTERFACE` block, you may get incorrect results unless you are very careful with implicit typing. If you want to include interfaces for all the routines in your program, you can:

  — include the file `iflport.f90` from the INCLUDE directory of your distribution
    or

  — add a `USE IFLPORT` statement to access the `INTERFACE`s for all portability functions.

2-1

# ACCESS

*Determines file access values*

### Prototype

```
USE IFLPORT
```

or

```
INTEGER(4) FUNCTION ACCESS(NAME,MODE)
  CHARACTER(LEN=*) NAME,MODE
END FUNCTION ACCESS
```

NAME          Input. `CHARACTER(LEN=*)`. Name of the file whose accessibility is to be determined.

MODE          Input. `CHARACTER(LEN=*)`. Modes of accessibility to check for. `MODE` is a character string of length one or greater containing only the characters "r", "w", "x", or "" (a blank). These characters are interpreted as follows:

### Character Meaning

The characters within `MODE` can appear in any order.

### Usage

```
result = ACCESS (filename, mode)
```

### Results

The result is `INTEGER(4)`, and  is zero if all access permissions in `MODE` are true. If input values that you pass to the function are invalid, or if the file cannot be accessed in all of the modes specified, one of the following error codes is returned:

EACCES          Access denied;

EINVAL          The mode argument is invalid

ENOENT          File not found

These error values are derived from the possible values of ERRNO, in Microsoft* Visual C++*.

Symbolic values for these error codes are defined in `iflport.f90`.

The *filename* argument can contain either forward or backward slashes for path separators.

On Windows* operating systems, all files are readable. A test for read permission always returns 0.

### Example

```
OPEN(UNIT=1,FILE="IFLFILE.TXT",STATUS='UNKNOWN')
 WRITE (1,10) "THIS IS A TEST"
 CLOSE(UNIT=1)
! checks for read and write permission on the file
"IFLFILE.TXT"

J = ACCESS ("IFLFILE.TXT", "rw")
PRINT *, J
! checks whether "IFLFILE.TXT" is executable. It is
not, since
! it does not end in .COM, .EXE, .BAT, or .CMD
J = ACCESS ("IFLFILE.TXT","x")
PRINT *, J
10 FORMAT(A)
END
```

# BEEPQQ

*Invokes the speaker*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
   SUBROUTINE BEEPQQ(FREQ,DUR)
      INTEGER(4) FREQ, DUR
 END SUBROUTINE
END INTERFACE
```

### Description

Invokes the speaker at the specified frequency for the specified duration in milliseconds.

### Usage

```
CALL BEEPQQ (FREQ, DUR)
```

FREQ            INTEGER(4). Frequency of the tone

DUR             Length of the tone in milliseconds.

BEEPQQ          does not return until the sound terminates.

### Example

```
USE IFLPORT
INTEGER(4) FREQ, DUR
FREQ = 4000
DUR  = 1000
CALL BEEPQQ(FREQ, DUR)
END
```

# BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN

*Compute the single-precision values of Bessel functions*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
        REAL(4) FUNCTION BESJ0(X)
        REAL(4) X
        END FUNCTION

        REAL(4) FUNCTION BESJ1(X)
        REAL(4) X
        END FUNCTION

        REAL(4) FUNCTION BESJN(N,X)
        INTEGER(4) N
        REAL(4) X
        END FUNCTION

        REAL(4) FUNCTION BESY0(X)
        REAL(4) X
        END FUNCTION

        REAL(4) FUNCTION BESY1(X)
        REAL(4) X
        END FUNCTION

        REAL(4) FUNCTION BESYN(N,X)
        INTEGER(4) N
        REAL(4) X
        END FUNCTION
   END INTERFACE
```

### Description

Compute the single-precision values of Bessel functions of the first and second kinds.

### Usage

```
result = BESJ0 (REALPOSITIVE)
result = BESJ1 (REALPOSITIVE)
```

```
result = BESJN (n, REALPOSITIVE)
result = BESY0 (REALPOSITIVE)
result = BESY1 (REALPOSITIVE)
result = BESYN (n, REALPOSITIVE)
```

REALPOSITIVE  REAL(4). Independent variable for a Bessel function. Must be greater than or equal to zero.

N             INTEGER(4) unless changed by the user. Specifies the order of the selected Bessel function computation.

### Results

BESJ0, BESJ1, and BESJN return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable REALPOSITIVE.

BESY0, BESY1, and BESYN return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable REALPOSITIVE.

Negative arguments return QNAN.

## CDFLOAT

*Converts an COMPLEX(4) to a DOUBLE PRECISION type*

### Prototype

```
INTERFACE
  DOUBLE PRECISION (REAL(8)) FUNCTION CDFLOAT (INPUT)
  COMPLEX(4), INTENT(IN)::INPUT
  END FUNCTION CDFLOAT
END INTERFACE
```

INPUT           a COMPLEX (KIND=4) value

### Description

CDFLOAT is an elemental function that converts a COMPLEX(KIND=4) type to DOUBLE PRECISION(REAL(8)).

### Output

The COMPLEX value converted to DOUBLE PRECISION.

# CHANGEDIRQQ

*Sets specified directory to the current directory*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
    LOGICAL(4) FUNCTION CHANGEDIRQQ(DIR)
        CHARACTER(LEN=*) DIR
    END FUNCTION
END INTERFACE
```

### Description

Sets the specified directory to the current, default directory.

### Usage

```
result = CHANGEDIRQQ (DIR)
```

DIR             CHARACTER(LEN=*). Directory to be made the current directory.

### Results

LOGICAL(4). .TRUE. if successful; otherwise, .FALSE..

If you do not specify a drive in the `DIR` string, the named directory on the current drive becomes the current directory. If you specify a drive in `DIR`, the named directory on the specified drive becomes the current directory.

### Example

```
USE IFLPORT
LOGICAL(4) CHANGEDIT
CHANGEDIT = CHANGEDIRQQ('c:\users')
END
```

# CHANGEDRIVEQQ

*Sets default drive*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
    LOGICAL(4) FUNCTION CHANGEDRIVEQQ(DriveName)
        CHARACTER (LEN=*) DriveName
    END FUNCTION
END INTERFACE
```

### Description

Sets the specified drive to the current, default drive.

### Usage

```
result = CHANGEDRIVEQQ (DriveName)
```

DriveName       `CHARACTER(LEN=*)`. `CHARACTER` value beginning with the drive letter.

### Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Drives are identified by a single alphabetic character. CHANGEDRIVEQQ examines only the first character of DriveName. The drive letter can be uppercase or lowercase.

CHANGEDRIVEQQ changes only the current drive. The current directory on the specified drive becomes the new current directory. If no current directory has been established on that drive, the root directory of the specified drive becomes the new current directory.

### Example

```
USE IFLPORT
LOGICAL(4) CHANGEDIT
CHANGEDIT = CHANGEDRIVEQQ('d')
IF (CHANGEDIT) THEN
    PRINT *,"CHANGEDRIVEQQ SUCCESSFUL"
ELSE
    PRINT *,"Drive could not be changed"
ENDIF
END
```

# CHDIR

*Changes the default directory.*

### Prototype

```
USE IFLPORT
```

Or

```
INTERFACE
INTEGER(4) FUNCTION CHDIR(DIRECTORY_NAME)
        CHARACTER(LEN=*) DIRECTORY_NAME
```

```
 END FUNCTION CHDIR
END INTERFACE
```

## Usage

```
result = CHDIR(NEW_DIRECTORY)
```

NEW_DIRECTORY `CHARACTER(LEN=*)`. Name of directory to become the default directory.

## Results

The result type is `INTEGER(4)`. `CHDIR` returns zero if the directory was changed successfully; otherwise, an error code. Possible error codes are:

ENOENT   The named directory does not exist.

ENOTDIR   The `NEW_DIRECTORY` parameter is not a directory.

## Example

```
USE IFLPORT
CHARACTER(LEN=16) NEW_DIRECTORY
LOGICAL(4) CHANGEDIT
NEW_DIRECTORY="c:\program files"
CHANGEDIT=CHDIR(NEW_DIRECTORY)
IF (CHANGEDIT)  THEN
    PRINT *, "CHDIR SUCCESSFUL"
ELSE
    PRINT *,"Directory could not be changed"
ENDIF
END
```

# CLOCK

*Returns current time*

### Prototype

```
INTERFACE
  SUBROUTINE CLOCK (TIMESTR)
  CHARACTER TIMESTR(LEN=8)
  END SUBROUTINE CLOCK
END INTERFACE
```

TIMESTR      the current time in the form of hh:mm:ss, using a
24-hour clock. Must be a CHARACTER variable,
STRUCTURE component, or ARRAY element at least
eight characters long.

### Description

This function returns the current time.

### Output

The current time in the form of hh:mm:ss, using a 24-hour clock.

# CLOCKX

*Returns processor clock*

### Prototype

```
INTERFACE
  SUBROUTINE CLOCKX (CLOCK)
  REAL(8) CLOCK
  END SUBROUTINE CLOCKX
```

```
END INTERFACE
```

CLOCK             current time

### Description

This function returns the processor clock to the nearest microsecond.

### Output

Processor clock to the nearest microsecond.

# COMMITQQ

*Forces execution of any pending write operation(s)*

### Prototype

```
USE IFLPORT
```

Or

```
INTERFACE
LOGICAL(4) FUNCTION COMMITQQ(LUN)
          INTEGER(4) LUN
      END FUNCTION COMMITQQ
END INTERFACE
```

### Description

Forces the operating system to execute any pending write operation(s) for the file associated with a specified unit to the file's physical device.  Same functionality as FLUSH.

### Usage

```
result = COMMITQQ (LUN)
```

LUN                     Input. `INTEGER(4)`. Fortran logical unit (`LUN`)
                        attached to a file to be flushed from cache memory to a
                        physical device.

### Results

The result type is `LOGICAL(4)`. If an open `LUN` number is supplied,
`.TRUE.` is returned and uncommitted records (if any) are written. If an
unopened `LUN` is supplied, `.FALSE.` is returned.

Data written to files is often written into buffers, and held until the buffer is
full. Only when the buffer is full, is the data written to the device. Data in
the buffer is automatically flushed to disk when the file is closed. However,
if the program or the computer crashes before the data is transferred from
buffers, the data can be lost.

`COMMITQQ` forces any cached data intended for a file on a physical device
to be written to that device immediately. This is called flushing the file.

### Example

```
USE IFLPORT
INTEGER LUN / 10 /
INTEGER len
CHARACTER(80) stuff
OPEN(LUN, FILE='COMMITQQ.TST', ACCESS='Sequential')
DO WHILE (.TRUE.)
  WRITE (*, '(A, \)') 'Enter some data (Hit RETURN to
exit): '
  len = GETSTRQQ (stuff)
  IF (len .EQ. 0) EXIT
  WRITE (LUN, *) stuff
  IF (.NOT. COMMITQQ(LUN)) WRITE (*,*) 'Failed'
END DO
CLOSE (LUN)
END
```

## COMPL

*Returns a BIT-WISE Complement or*
*logical .NOT. of the input value*

### Prototype

```
INTERFACE
  LOGICAL FUNCTION COMPL(INVAL)
  LOGICAL INVAL
  END LOGICAL FUNCTION COMPL
END INTERFACE
```

### Description

Performs a bitwise Complement for `INTEGER` input or equivalent of
`.NOT.(IN)` for logical input.

### Output

If the input is logical, the result is logical.Otherwise, the result is Boolean
- a Cray bitset. With a Boolean result, use a BIT-WISE complement. For the
logical compl, just toggle 1<-->0.

## CTIME

*Converts a given time into a*
*24- character ASCII string*

### Prototype

```
INTERFACE
    CHARACTER(LEN=24) FUNCTION CTIME(TIME)
    INTEGER TIME
```

```
    END FUNCTION CTIME
END INTERFACE
```

TIME             elapsed time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

### Description

This function takes an `INTEGER`(4) variable or expression as input. The input value is some number of seconds since midnight of January 1, 1970, in Greenwich Mean Time. `CTIME` converts this integer input value into an ASCII character string.

### Output

A 24-character ASCII string in the form: Thu Jan 15 00:00:01 1970.

# DATE

*Returns the current date and time as ASCII string*

```
INTERFACE
    SUBROUTINE DATE(DATESTR)
    CHARACTER(LEN=9) DATESTR
    END SUBROUTINE DATE
END INTERFACE
```

### Description

This function returns the current date and time as a nine character ASCII string.

### Output

A 9-character ASCII string in the form: `dd-mmm-yy` where:

`dd`         is the 2-digit date

| mmm | is the 3 letter month |
|-----|-----------------------|
| yy  | is the last two digits of the year |

> **WARNING.** *This routine may cause problems with the year 2000. Use DATE_AND_TIME or DATE4 instead.*

### Example

The following program gets and prints the current system date. Its output could be, for example, "12-Jul-96" (without the quotes).

```fortran
PROGRAM datetest
  CHARACTER(9) :: today
  INTRINSIC DATE
  CALL DATE(today)
  PRINT *, today
END
```

## DATE4

*Returns the current date and time as ASCII string*

### Prototype

```fortran
INTERFACE
    SUBROUTINE DATE4(DATESTR)
    CHARACTER(LEN=11) DATESTR
    END SUBROUTINE DATE4
END INTERFACE
```

### Description

This function returns the current date and time as an eleven character ASCII string.

### Output

An 11-character ASCII string in the form: `dd-mmm-yyyy` where:

`dd`                    is the 2-digit date

`mmm`                   is the 3 letter month

`yy`                    is the last two digits of the year

> **NOTE.** *Although using* `DATE_AND_TIME` *is better, since it is now standard in the Fortran language, this routine gives you the functionality of the old DATE routine and is year-2000 compliant.*

# DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN

*Compute the double-precision numbers*
*of Bessel functions.*

### Prototype

```
USE IFLPORT or
INTERFACE
        REAL(8) FUNCTION DBESJ0(X)
        REAL(8) X
        END FUNCTION
        REAL(8) FUNCTION DBESJ1(X)
        REAL(8) X
```

```
                        END FUNCTION

                        REAL(8) FUNCTION DBESJN(N,X)
                        INTEGER(4) N
                        REAL(8) X
                        END FUNCTION

                        REAL(8) FUNCTION DBESY0(X)
                        REAL(8) X
                        END FUNCTION

                        REAL(8) FUNCTION DBESY1(X)
                        REAL(8) X
                        END FUNCTION
                        REAL(8) FUNCTION DBESYN(N,X)
                        INTEGER(4) N
                        REAL(8) X
                        END FUNCTION
                  END INTERFACE
```

### Descriptions

Compute the double-precision values of Bessel functions of the first and second kinds.

### Usage

```
result = DBESJ0 (DOUBLEPOS)
result = DBESJ1 (DOUBLEPOS)
result = DBESJN (n, DOUBLEPOS)
result = DBESY0 (DOUBLEPOS)
result = DBESY1 (DOUBLEPOS)
result = DBESYN (n, DOUBLEPOS)
```

DOUBLEPOS        REAL(8). Independent variable for a Bessel function. Must be greater than or equal to zero.

N                    `Integer`. Specifies the order of the selected Bessel function computation.

### Results

`DBESJ0`, `DBESJ1`, and `DBESJN` return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable `DOUBLEPOSe`.

`DBESY0`, `DBESY1`, and `DBESYN` return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable `DOUBLEPOSe`.

Negative arguments cause `DBESY0`, `DBESY1`, and `DBESYN` to return a huge negative value.

### Example

```
    USE IFLPORT
    REAL(8) BESNUM, BESOUT
10  READ *, BESNUM
    BESOUT = DBESJ0(BESNUM)
    PRINT *, 'Result is ',BESOUT
    GOTO 10
    END
```

# DCLOCK

*Provides elapsed time in seconds since
the start of the current process.*

### Prototype

```
INTERFACE
    REAL(8) FUNCTION DCLOCK()
    END FUNCTION
END INTERFACE
```

### Description

This function returns the elapsed time since the start of your process as a `DOUBLE PRECISION` number.

**Note:** The first call to `DCLOCK` performs calibration.

### Class

Elemental nonstandard function.

### Result Type and Type Parameter

`DOUBLE PRECISION`

### Output

The time in seconds since the beginning of your process. This routine provides accurate timing to the nearest microsecond, taking into account the frequency of the processor where the current process is running. You can obtain equivalent results using standard Fortran by using the CPU_TIME intrinsic function.

### Examples

```
DOUBLE PRECISION START_TIME, STOP_TIME, DCLOCK
EXTERNAL DCLOCK
START_CLOCK = DCLOCK()
CALL FOO()
STOP_CLOCK = DCLOCK()
PRINT *, "foo took:", STOP_CLOCK - START_CLOCK, "seconds."
```

# DELDIRQQ

*Deletes a specified directory*

### Prototype

```
USE IFLPORT
```

or

```
        INTERFACE
         LOGICAL(4) FUNCTION DELDIRQQ(DirName)
              CHARACTER(LEN=*) DirName
          END FUNCTION
        END INTERFACE
```

### Usage

`result = DELDIRQQ (DIRNAME)`

DIRNAME          `CHARACTER(LEN=*)`. `CHARACTER` value containing
                 the name of the directory to be deleted.

### Results

The result is `LOGICAL(4)`. The result is `.TRUE.` if successful; otherwise,
`.FALSE.`.

The directory to be deleted must be empty. It cannot be the current
directory, the root directory, or a directory currently in use by another
process.

# DELFILESQQ

*Deletes files matching specification*

### Prototype

```
USE IFLPORT

or

INTERFACE
  INTEGER(4) FUNCTION DELFILESQQ(FILESPEC)
    CHARACTER(LEN=*) FILESPEC
  END FUNCTION
END INTERFACE
```

### Description

Deletes all files matching the name specification, which can contain wildcards (* and ?).

### Usage

```
result = DELFILESQQ (FILESPEC)
```

FILESPEC      `CHARACTER(LEN=*)`. File(s) to be deleted. Can contain wildcards (* and ?).

### Results

The result type is `INTEGER(2)`. The return value is the number of files deleted.

You can use wildcards to delete more than one file at a time. `DELFILESQQ` does not delete directories or system, hidden, or read-only files. Use this function with caution because it can delete many files at once. If a file is in use by another process (for example, if it is open in another process), it cannot be deleted.

### Example

```
USE IFLPORT
INTEGER(4) len, count
CHARACTER(80) file
CHARACTER(1) ch
WRITE(*,*) "Enter names of files to delete: "
len = GETSTRQQ(file)
IF (file(1:len) .EQ. '*.*') THEN
   WRITE(*,*) "Are you sure (Y/N)?"
   ch = GETCHARQQ()
   IF ((ch .NE. 'Y') .AND. (ch .NE. 'y')) THEN
        STOP "No files deleted"
   ELSE
       PRINT *,"OK, deleting all files"
   ENDIF
END IF
count = DELFILESQQ(file)
```

```
WRITE(*,*) "Deleted ", count, " files."
END
```

# DFLOATI

*Converts an INTEGER(2) to a DOUBLE
PRECISION type*

### Prototype

```
INTERFACE
  DOUBLE PRECISION (REAL(8)) FUNCTION DFLOATI (INPUT)
  INTEGER(2), INTENT(IN)::INPUT
  END FUNCTION DFLOATI
END INTERFACE
```

INPUT            a scalar INTEGER (KIND=2) value

### Description

DFLOATI is an elemental function that converts a scalar integer (KIND=2)
type to DOUBLE  PRECISION (REAL(8)).

### Output

The integer value converted to DOUBLE  PRECISION.

## DFLOATJ

*Converts an INTEGER(4) to a DOUBLE
PRECISION type*

### Prototype

```
INTERFACE
  DOUBLE PRECISION (REAL(8)) FUNCTION DFLOATJ (INPUT)
  INTEGER(4), INTENT(IN)::INPUT
  END FUNCTION DFLOATJ
END INTERFACE
INPUT            an INTEGER(4) value or expression
```

### Description

DFLOATJ is an elemental function that converts an INTEGER(4) type to
DOUBLE PRECISION(REAL(8)) type.

### Output

The integer value converted to DOUBLE PRECISION.

## DFLOATK

*Converts an INTEGER(8) type to a
DOUBLE PRECISION type*

### Prototype

```
INTERFACE
  REAL(8) FUNCTION DFLOATK (INPUT)
  INTEGER(8), INTENT(IN)::INPUT
  END FUNCTION DFLOATJ
```

```
END INTERFACE
```

INPUT            an INTEGER(8) value or expression.

### Description

DFLOATK is an elemental function that converts an INTEGER(8) type to a DOUBLE PRECISION (or REAL(8)) type.

### Output

The integer value converted to DOUBLE PRECISION.

# DRAND

*Generates successive pseudorandom numbers in the range of 0. to 1.*

### Prototype

```
INTERFACE
    REAL(8) FUNCTION RAND()
    END FUNCTION RAND
END INTERFACE
```

### Description

Generate successive pseudorandom numbers uniformly distributed in the range of 0.0 to 1.0.

### Class

Elemental nonstandard function.

### Result Type and Type Parameter

REAL(8) type.

**Example**

```
REAL(8) rv
rv = RAND()
```

**NOTE.** *For details about restarting the pseudorandom number generator used by* IRAND *and* RAND, *see the* "SRAND" *section.*

# DRANSET

*Sets the seed for RANGET*

**Prototype**

```
INTERFACE
   SUBROUTINE DRANSET (ISEED)
   REAL(8) ISEED
   END SUBROUTINE DRANSET
END INTERFACE
```

**Description**

Allows you to set the seed for RANGET, changing the sequence of pseudo-random numbers.

**Output**

Changes the internal value of the random number generator seed for RANGET.

# DSHIFTL

*Double shift left*

### Prototype

```
INTERFACE
  INTEGER(8) FUNCTION DSHIFTL (LEFT, RIGHT, SHIFT)
  INTEGER(8) LEFT, RIGHT, SHIFT
  END FUNCTION DSHIFTL
END INTERFACE
```

LEFT             a 64-bit integer expression.

RIGHT            a 64-bit integer expression.

SHIFT            shift count.

### Description

This function takes two 64-bit values and a shift count, and returns a 64-bit value comprised of the 64 bits starting at 64-SHIFT in left continuing through to right.

### Example

```
      INTERFACE
       INTEGER(8) FUNCTION DSHIFTL(LEFT,RIGHT,SHIFT)
        INTEGER(8) LEFT,RIGHT,SHIFT
        END FUNCTION DSHIFTL
       END INTERFACE

        INTEGER(8) LEFT/Z'111122221111222'/
        INTEGER(8) RIGHT/Z'FFFFFFFFFFFFFF'/
        PRINT *, DSHIFTL(LEFT, RIGHT,16_8)
      END
```

The correct output for this example is: 1306643199093243919.

**Output**

This function performs a shift left and extract, for compatibility with CRAY Fortran code.

# DSHIFTR

*Double shift right*

### Prototype

```
INTERFACE
    INTEGER(8) FUNCTION DSHIFTR(ILEFT,IRIGHT,ISHIFT)
    INTEGER(8) ILEFT,IRIGHT
    INTEGER(4) ISHFIT
    END FUNCTION DSHIFTR
END INTERFACE
```

ILEFT            a 64-bit integer value.

IRIGHT           a 64-bit integer value.

ISHIFT           an integer shift count.

### Description

Interprets ILEFT and IRIGHT as the upper and lower parts, respectively, of a 128-bit integer. The result is the 64-bit string beginning with the bit ISHIFT of the 128-bit integer. The arguments, ILEFT and IRIGHT, are not altered unless the function result is assigned to the same storage location as either ILEFT or IRIGHT. ISHIFT should be in the range 0 to 64, inclusive.

### Example

```
        INTERFACE
            INTEGER(8) FUNCTION
DSHIFTR(LEFT,RIGHT,SHIFT)
            INTEGER(8) LEFT,RIGHT,SHIFT
           END FUNCTION DSHIFTR
        END INTERFACE
```

```
            INTEGER(8) LEFT/Z'111122221111222'/
            INTEGER(8) RIGHT/Z'FFFFFFFFFFFFF'/
            PRINT *, DSHIFTR(LEFT, RIGHT,16_8)
            END
```

The correct output for this example is: 1306606910610341887.

### Output

A single `INTEGER(8)` value.

# DTIME

*Returns the elapsed CPU time since the start of execution or the last call to DTIME.*

### Prototype

```
INTERFACE
  REAL(4) FUNCTION DTIME (TARRAY)
  REAL(4) TARRAY(2)
  END FUNCTIOMN DTIME
END INTERFACE
```

### Description

On the first call during the execution of a program, `DTIME` returns the time since the beginning of the program execution in the elements of `TARRAY`.

`TARRAY(1)` contains the user time, and `TARRAY(2)` contains the system time. The library routine makes use of the GetProcessTimes system library call. Subsequent calls to `DTIME` return the elapsed user and system time in `TARRAY` since the last call to `DTIME`.

**Output**

The function returns -1 for an error. The times are returned as elements of TARRAY, as described above. Times are in seconds. For the most accurate timing of your process, use the standard Fortran 95 intrinsic CPU_TIME.

# ETIME

*Returns the elapsed time in seconds*
*since the start of execution*

**Prototype**

```
INTERFACE
    REAL(4) FUNCTION ETIME(TARRAY)
    REAL(4) TARRAY(2)
    END FUNCTION
END INTERFACE
```

**Description**

This function returns the elapsed time in seconds since the beginning of execution of the current process. The accuracy of ETIME is limited to the accuracy of the GetProcessTimes system call, approximately 1/100th of a second. For the most accurate timing of your routine, use the Fortran 95 standard intrinsic CPU_TIME.

**Output**

The function returns -1 in case of an error. TARRAY(1) contains the user time in seconds since the start of the process. The system time in seconds since the start of the process is returned in TARRAY(2).

# EXIT

*Closes all files and terminates the
program*

### Prototype

```
INTERFACE
  FUNCTION EXIT (STATUS)
  INTEGER ISEED
  END FUNCTIOMN EXIT
END INTERFACE
```

### Optional Argument

STATUS

### Description

Close all files and terminate the program.

### Class

Nonstandard subroutine.

### Argument

If STATUS is supplied, the calling program exits with a return code status of STATUS. Otherwise the return code status is indeterminate.

In csh the $status environment variable holds the return code for the last executed command. In ksh, the $? environment variable holds the return code.

### Example

The following program exits before the second PRINT statement.

```
PROGRAM testexit
 INTEGER stat
 PRINT *, "Program prints this line."
```

```
 stat = 3
 CALL EXIT(stat)
END
```

This program produces the following output:

```
Program prints this line.
```

The return code is saved in the `$status` or `$?` environment variable; it is not printed.

# FDATE

*Returns the current date and time*

### Prototype

```
INTERFACE
    SUBROUTINE FDATE(STRING)
    CHARACTER(LEN=24) STRING
    END SUBROUTINE FDATE
END INTERFACE
```

STRING          a 24-byte character variable or array element in which the result of FDATE is stored.

### Description

This routine returns the current date and time in STRING. Any value in STRING before the call is destroyed.

### Output

A 24-character string with the form: Thu Jan 15 00:00:01 1970.

# FGETC

*Reads one byte from a file*

### Prototype

```
INTERFACE
    INTEGER FUNCTION FGETC(LUNIT,NCHAR)
    INTEGER LUNIT
    CHARACTER(LEN=1) NCHAR
    END FUNCTION FGETC
END INTERFACE
```

LUNIT            a logical unit number

NCHAR            a character variable

### Description

This routine reads one byte from a file at its current position. If there is an error during the read, the I/O error number is returned. Otherwise, for a successful read, FGETC returns zero. You should be aware that for unformatted files, special bytes such as a record length indicator are present on each record, and may require special handling when using this function. In general, record length indicators for unformatted files are the first four bytes of each record. The logical unit number must be in the range from 0 to 100, and must be currently connected to a file when FGETC is called.

This routine is thread-safe, and locks the associated stream before I/O is performed.

### Output

A zero status is returned if successful, non-zero if failure. NCHAR is assigned the next sequential byte that would be read from the file connected to LUNIT.

## FINDFILEQQ

*Searches for a specified file*

### Prototype

```
USE IFLPORT
Or
NTERFACE
 IINTEGER(4) FUNCTION FINDFILEQQ(FILE, ENV, BUF)
   CHARACTER(LEN=*) FILE, ENV, BUF
 END FUNCTION
END INTERFACE
```

### Description

Searches for a specified file in the directories listed in the path contained in the environment variable.

### Usage

```
result = FINDFILEQQ (file, env, buf)
```

| | |
|---|---|
| FILE | CHARACTER(LEN=*). Name of the file to be found. |
| ENV | CHARACTER(LEN=*). Name of an environment variable containing the path to be searched. |
| BUF | CHARACTER(LEN=*). Buffer to receive the full path of the file found. |

### Results

The result type is INTEGER(4). The result is the length of the characters containing the full path of the found file returned in BUF, or 0 if no file is found.

### Example

```
USE IFLPORT
CHARACTER(256) BUF
```

```
CHARACTER(20) FILE,ENV
INTEGER(4) result
FILE = "libc.lib"
ENV = "LIB"
result = FINDFILEQQ(FILE,ENV, buf)
WRITE (*,*) BUF
END
```

# FLOATI

*Converts an intput value to a real value*

### Prototype

```
INTERFACE
   REAL(4) FUNCTION FLOATI (IN)
   INTEGER(2) IN
   END FUNCTION FLOATI
END INTERFACE
```

IN                an INTEGER(2) expression

### Description

Converts an input value to REAL(4).

### Output

The equivalent REAL(4) value.

# FLOATJ

*Converts an integer to a real value*

### Prototype

```
INTERFACE
  REAL(4) FUNCTION FLOATJ (IN)
  INTEGER(4) IN
  END FUNCTION FLOATI
END INTERFACE
```

IN    an INTEGER(4) expression

### Description

Converts an input value to REAL(4).

### Output

The equivalent REAL(4) value.

# FLUSH

*Flushes contents of file buffer to an
external file*

### Prototype

```
INTERFACE
    SUBROUTINE FLUSH(LUNIT)
    INTEGER LUNIT
    END SUBROUTINE  FLUSH
END INTERFACE
```

LUNIT               a logical unit number

### Description

This routine flushes the contents of the file buffer to the external file, forcing an immediate write. This is most useful for files that are connected to a terminal. The logical unit number must be in the range from 0 to 100, and must be currently connected to a file when flush is called. This routine is thread-safe, and locks the associated stream before I/O is performed.

### Output

ERRNO is set on failure.

# FOR_CHECK_FLAWED_PENTIUM

*Checks the processor*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
  SUBROUTINE FOR_CHECK_FLAWED_PENTIUM
  END SUBROUTINE
END INTERFACE
```

### Description

Checks the processor to determine if it shows characteristics of the Pentium® floating-point divide flaw.

It is invoked for a Fortran program if you compile with the /Qfdiv compiler switch.

### Usage

```
result = FOR_CHECK_FLAWED_PENTIUM ( )
```

### Results

If the floating-point divide flaw is found, an error message is displayed and the calling program is terminated.

You can bypass this action by setting environment variable FOR_RUN_FLAWED_PENTIUM to the value of .TRUE..

### Example

```
USE IFLPORT
REAL*8 X, Y, Z
X = 5244795.0
Y = 3932159.0
Z = X - (X/Y) * Y
IF (Z .NE. 0) THEN          ! If flawed, Z will be 256
    PRINT *, " FDIV flaw detected on Pentium"
ENDIF
END
```

# FOR_GET_FPE

*Rettrun the current settings of the floating-point exception flags*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
  INTEGER(4) FUNCTION FOR_GET_FPE()
   END FUNCTION
END INTERFACE
```

### Description

Returns the current settings of floating-point exception flags. This routine can be called from a C or Fortran program.

### Usage

```
result = FOR_GET_FPE ( )
```

### Results

The result type is INTEGER(4). The return value represents the settings of the current processor floating-point exception flags. The meanings of the bits are defined in the IFLPORT module file.

### Example

```
USE IFLPORT
INTEGER(4) FPE_FLAGS
FPE_FLAGS = FOR_GET_FPE ( )
END
```

# FPUTC

*Writes a character a a file*

### Prototype

```
INTERFACE
    INTEGER FUNCTION FPUTC(LUNIT, CH)
    INTEGER LUNIT
    CHARACTER(LEN=1) CH
    END FUNCTION FPUTC
END INTERFACE
```

LUNIT            unit number of a file.

CH               a character variable.

### Description

Writes a character to the file specified by the Fortran external unit.

### Output

Zero if successful. Otherwise, an error code is returned.

# FREE

*Frees a block of memory*

### Prototype

```
INTERFACE
    SUBROUTINE FREE(ADDRESS)
    INTEGER ADDRESS
    END FUNCTION FREE
END INTERFACE
```

ADDRESS          starting address of the memory block to be freed.

### Description

Frees a block of currently allocated memory.

**NOTE.**  *This should only be used for pointers that were previously allocated by* MALLOC.

# FSEEK

*Positions a file from a specified seek point*

## Prototype

```
INTERFACE
    SUBROUTINE FSEEK(LUNIT,OFFSET,FROM)
    INTEGER LUNIT,OFFSET,FROM
    END SUBROUTINE FSEEK
END INTERFACE
```

| | |
|---|---|
| LUNIT | a logical unit number. |
| OFFSET | an integer expression, whose value denotes an offset in bytes from the seek point FROM. |
| FROM | an integer expression, whose value must be 0, 1, or 2. |

    0    means the seek point is at the beginning of the file.

    1    means the seek point is on the current file position.

    2    means the seek point is at the end of the file.

## Description

This routine positions a file OFFSET bytes from the seek point given by the FROM parameter. You should be aware that for unformatted files, special bytes such as a record length indicator are present on each record, and may require special handling when using this function. In general, record length indicators for unformatted files are the first four bytes of each record. The logical unit number must be in the range from 0 to 100, and must be currently connected to a file when fseek is called.

This routine is thread-safe, and locks the associated stream before I/O is performed.

**Output**

A zero status is returned if successful; non-zero for failure.

# FOR_SET_FPE

*Sets the floating-point exception flags.*

### Prototype

```
USE IFLPORT
```
or
```
INTERFACE
  INTEGER(4) FUNCTION FOR_SET_FPE(EnableMask)
      INTEGER(4) EnableMask
  END FUNCTION
END INTERFACE
```

### Usage

```
result = FOR_SET_FPE(EnableMask)
```

EnableMask      Must be of type `INTEGER(4)`. It contains bit flags
                controlling floating-point exceptions.

### Results

The result type is `INTEGER(4)`. The return value represents the previous
settings of the floating-point exception flags. The meanings of the bits are
defined in the IFLPORT.f90 module file.

### Example

```
USE IFLPORT
INTEGER(4) OLD_FLAGS, NEW_FLAGS
OLD_FLAGS = FOR_SET_FPE (NEW_FLAGS)
END
```

# FOR_SET_REENTRANCY

*Controls the type of reentrancy locks on
the runtime library.*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
  INTEGER(4) FUNCTION FORr_SET_REENTRANCY(NEW_MODE)
      INTEGER(4) NEW_MODE
  END FUNCTION
END INTERFACE
```

### Usage

```
result = FOR_SET_REENTRANCY ( NEW_MODE )
```

NEW_MODE        INTEGER(4). Contains one of the following options:

FOR_K_REENTRANCY_NONE

Tells the runtime to do simple locking around critical
sections of RTL code. You should  use this type of
protection  when the Fortran libraries will *not* be
reentered due to asynchronous system traps (ASTs) or
threads within the application.

FOR_K_REENTRANCY_ASYNCH

Tells the runtime to perform simple locking and disables
ASTs around critical sections of  library code. You
should use this type of protection when the application
contains AST handlers that call the Fortran runtime
system.

FOR_K_REENTRANCY_THREADED

Tells the runtime  to perform thread locking. You should
use this type of protection in multithreaded applications.

FOR_K_REENTRANCY_INFO

> Queries the FORTRAN runtime system for the current level of reentrancy protection, and returns the result in NEW_MODE.

## Results

FOR_SET_REENTRANCY returns INTEGER(4). The return value tells you the previous setting of reentrancy NEW_MODE, unless the argument is FOR_K_REENTRANCY_INFO, in which case the return value represents the current setting.

You must link to a set of runtime libraries that support the level of reentrancy you desire. For example, FOR_SET_REENTRANCY ignores a request for thread protection (FOR_K_REENTRANCY_THREADED) if you do not compile your program with /MT.

## Example

```
PROGRAM SETREENT
  USE IFLPORT
  INTEGER(4) MODE
  CHARACTER*10 REENT_TXT(3) /'NONE  ', 'ASYNCH  ',
                             'THREADED'/
  INTEGER(4) P1, P2
  P1 = FOR_K_REENTRANCY_NONE
  P2 = FOR_K_REENTRANCY_INFO
  PRINT*,'Setting Reentrancy mode to '
                 ,REENT_TXT(MODE+1)'
  MODE = FOR_SET_REENTRANCY(P1)
  PRINT*,'Previous Reentrancy mode was  '
             ,REENT_TXT(MODE+1)
  MODE = FOR_SET_REENTRANCY(P2)
  PRINT*,'Current Reentrancy mode is '
             ,REENT_TXT(MODE+1)
  END
```

# FTELL

*Returns the file position of a file*

### Prototype

```
INTERFACE
    INTEGER(4) FUNCTION FTELL(LUNIT)
    END FUNCTION FTELL
END INTERFACE
```

`LUNIT`            a logical unit number

### Description

This routine returns the file position of the file connected to the input logical unit. The file position is the number of bytes from the beginning of the file.

You should be aware that for unformatted files, special bytes such as a record length indicator are present on each record, and are counted when using this function. In general, record length indicators for unformatted files are the first four bytes of each record.

The logical unit number must be in the range from 0 to 100, and must be currently connected to a file when `ftell` is called.

This routine is thread-safe, and locks the associated stream before I/O is performed.

### Output

`ERRNO` is set on failure.

# FULLPATHQQ

*Returns the full path for a specified file
or directory.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  INTEGER(4) FUNCTION FULLPATHQQ(NAME, FULLPATH)
    CHARACTER(LEN=*) NAME, FULLPATH
  END FUNCTION
END INTERFACE
```

### Usage

```
result = FULLPATHQQ ( name, fullpath )
```

NAME            `CHARACTER(LEN=*)`. File name for which you want a full path. The file can be the name of a file in the current directory, or a relative directory or filename.

FULLPATH        `CHARACTER(LEN=*)`. `CHARACTER` VALUE that receives the full path of the item specified in `NAME`.

### Results

The result is the length of the full pathname in `CHARACTER`s, or 0 if the function fails. The function may fail if the name supplied is not an existing file. The length of `FULLPATH` will vary, depending on how deeply the directories are nested on the drive you are using. If the full path is longer than the character buffer provided to return it (`FULLPATH`), `FULLPATH`QQ returns only that portion of the path that fits.

You should verify the length of the path before using the value returned in `FULLPATH`. If the longest full path you are likely to encounter does not fit into the buffer you are using, allocate a larger character buffer. You can allocate the largest possible path buffer with the following statements:

```
USE IFLPORT
CHARACTER(MAXPATH) FULLPATH
```

MAXPATH is a symbolic constant defined in module `IFLPORT.F90` as 260.

### Example

```
USE IFLPORT
CHARACTER(MAXPATH) BUF
CHARACTER(3)        DRIVE
CHARACTER(256)      DIR
CHARACTER(256)      NAME
CHARACTER(256)      EXT
CHARACTER(256)      FILE
INTEGER(4)          LEN


DO WHILE (.TRUE.)
    WRITE (*,*)
    WRITE (*,'(A)') ' Enter filename (Hit RETURN to
                        exit): '
    LEN = GETSTRQQ(FILE)
     IF (LEN .EQ. 0) EXIT
     LEN = FULLPATHQQ(FILE, BUF)
     IF (LEN .GT. 0) THEN
       WRITE (*,*) buf(:len)
     ELSE
       WRITE (*,*) 'Can''t get full path'
       EXIT
     END IF
    !
!Split path
 WRITE (*,*)
 LEN = SPLITPATHQQ(BUF, DRIVE, DIR, NAME, EXT)
 IF (LEN .NE. 0) THEN
   WRITE (*, 900) ' Drive: ', DRIVE
   WRITE (*, 900) ' Directory: ', DIR(1:LEN)
   WRITE (*, 900) ' Name: ', NAME
```

```
        WRITE (*, 900) ' Extension: ', EXT
      ELSE
        WRITE (*, *) 'Can''t split path'
      END IF
    END DO
900  FORMAT (A, A)
    END
```

# GERRNO

*Returns a message for last error*

### Prototype

```
INTERFACE
    SUBROUTINE GERRNO(ERRORMSG)
    CHARACTER(LEN=*) ERRORMSG
    END SUBROUTINE
END INTERFACE
```

STRING          message for the last error detected

### Description

This routine returns a message for the last error detected.

### Output

The last error detected in STRING.

# GETARG

*Gets a specified command-line argument*

## Prototype

```
INTERFACE
    SUBROUTINE GETARG (N, STRING)
    INTEGER N
    CHARACTER(LEN=*) STRING
    END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| N | an integer representing which command-line argument to get. The 0th command-line argument is the command name. |
| STRING | the space for argument to returned into. STRING must be large enough to hold the result or it will be truncated |

## Description

This subroutine gets Nth command-line argument.

If there are not at least I arguments on the command line for your application, an error message is generated, errno is set to a non-zero value (ERANGE), and control is returned to your application. GETARG will not abort your program if you ask for an argument that is not there.

## Output

A string representing the Nth command-line argument to the executing program.

### Example

The following code assigns `ARG` the value of the second argument passed to the program.

```
CHARACTER(10) ARG
CALL GETARG(2, ARG)
```

# GETCHARQQ

*Gets the next keystroke.*

### Prototype

```
USE IFLPORT
or
! Get character from console
 CHARACTER(LEN=1) FUNCTION GETCHARQQ()
 END FUNCTION GETCHARQQ
```

### Usage

```
result = GETCHARQQ ( )
```

### Results

The result type is `CHARACTER(1)`, and has the value of the key that was pressed. The value can be any ASCII character. If the key pressed is represented by a single ASCII character, `GETCHARQQ` returns the character. If the key pressed is a function or direction key, a hex #00 or #E0 is returned. If you need to know which function or direction was pressed, call `GETCHARQQ` a second time to get the extended code for the key.

If there is no keystroke waiting in the keyboard buffer, `GETCHARQQ` will wait indefinitely until there is one, and then returns it. To determine in advance whether there is a character in the keyborad buffer, use

PEEKCHARQQ, which returns .TRUE. if there is a character waiting in the
keyboard buffer, and .FALSE. if not. This can prevent a program from
hanging while GETCHARQQ waits for a keystroke that isn't there.

### Example

```
!  Program to demonstrate GETCHARQQ
USE IFLPORT
CHARACTER(1) key / 'A' /
PARAMETER (ESC = 27)
PARAMETER (NOREP = 0)
WRITE (*,*) ' Type a key: (or q to quit)'
!  Read keys until ESC or q is pressed
DO WHILE (ICHAR (key) .NE. ESC)
   key = GETCHARQQ()
!  Some extended keys have no ASCII representation
   IF(ICHAR(key) .EQ. NOREP) THEN
     key = GETCHARQQ()
     WRITE (*, 900) 'Not ASCII. Char = NA'
     WRITE (*,*)
!  Otherwise, there is only one key
   ELSE
     WRITE (*,900) 'ASCII. Char = '
     WRITE (*,901) key
   END IF
   IF (key .EQ. 'q' ) THEN
     EXIT
   END IF
   END DO
900   FORMAT (1X, A)
901   FORMAT (A)
END
```

# GETCONTROLFPQQ

*Returns the floating-point processor control word.*

## Prototype

```
USE IFLPORT
```

or

```
INTERFACE
  SUBROUTINE GETCONTROLFPQQ(CONTROL)
    INTEGER(2) CONTROL
  END SUBROUTINE
END INTERFACE
```

## Usage

```
CALL GETCONTROLFPQQ (CONTROL)
```

CONTROL          INTEGER(2). Floating-point processor control word.

The floating-point control word is a set of flags that determines various modes of the floating-point coprocessor. The IFLPORT.F90 module file contains constants defined for the control word as follows:

| Parameter Name | Hex Value | Description |
|---|---|---|
| FPCW$MCW_IC | Z'1000' | **Infinity control mask** |
| FPCW$AFFINE | Z'1000' | Affine infinity |
| FPCW$PROJECTIVE | Z'0000' | Projective infinity |
| FPCW$MCW_PC | Z'0300' | **Precision control mask** |
| FPCW$64 | Z'0300' | 64-bit precision |
| FPCW$53 | Z'0200' | 53-bit precision |
| FPCW$24 | Z'0000' | 24-bit precision |
| FPCW$MCW_RC | Z'0C00' | **Rounding control mask** |
| FPCW$CHOP | Z'0C00' | Truncate |

| Parameter Name | Hex Value | Description |
|---|---|---|
| FPCW$UP | Z'0800' | Round up |
| FPCW$DOWN | Z'0400' | Round down |
| FPCW$NEAR | Z'0000' | Round to nearest |
| FPCW$MSW_EM | Z'003F' | **Exception mask** |
| FPCW$INVALID | Z'0001' | Allow invalid numbers |
| FPCW$DENORMAL | Z'0002' | Allow denormals (very small numbers) |
| FPCW$ZERODIVIDE | Z'0004' | Allow divide by zero |
| FPCW$OVERFLOW | Z'0008' | Allow overflow |
| FPCW$UNDERFLOW | Z'0010' | Allow underflow |
| FPCW$INEXACT | Z'0020' | Allow inexact precision |

By default, the floating-point control word settings are 53-bit precision, round to nearest, and the DENORMAL, UNDERFLOW and INEXACT precision exceptions disabled. An exception is disabled if its flag is set to 1 and enabled if its flag is cleared to 0. You can disable exceptions by setting the flags to 1 with SETCONTROLFPQQ.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, the exception generates an appropriate special value (NaN or signed infinity), but the program continues. When printing out such a value, the runtime library represents a NaN as ?????, positive infinity as +++++++, and negative infinity as ----------.

You can find out which exceptions (if any) occurred by calling GETSTATUSFPQQ. If you have enabled errors on floating-point exceptions, clearing the flags to 0 with SETCONTROLFPQQ, an interrupt is generated when the exception occurs. Normally, these interrupts cause errors, but you can capture the interrupts with SIGNALQQ and branch to your own error-handling routines.

You can use GETCONTROLFPQQ to retrieve the current control word and SETCONTROLFPQQ to change the control word. In most cases, you will not need to change the default settings.

### Example

```
USE IFLPORT
INTEGER(2) CONTROL
CALL GETCONTROLFPQQ (CONTROL)
PRINT 10,CONTROL
10 FORMAT('Initial control settings ',Z)
      !if not rounding down
IF (IAND(CONTROL, FPCW$DOWN) .NE. FPCW$DOWN) THEN
  CONTROL = IAND(CONTROL, NOT(FPCW$MCW_RC))
                                  !clear all rounding
  CONTROL = IOR(control, FPCW$DOWN)
                                  !set to round down
  CALL SETCONTROLFPQQ(CONTROL)
CALL GETCONTROLFPQQ(CONTROL)
PRINT 20,control
20 FORMAT('Final control settings ',Z)
END IF
END
```

# GETCWD

*Retrieves the path of the current working directory.*

### Prototype

```
USE IFLPORT
```

or

```
   INTERFACE
      INTEGER(4) FUNCTION GETCWD(DIRECTORY)
          CHARACTER(LEN=*) DIRECTORY
      END FUNCTION GETCWD
   END INTERFACE
```

### Usage

```
result = GETCWD (directory)
```

DIRECTORY      CHARACTER (LEN=*). Character value that receives the current working directory path, including drive letter.

### Results

GETCWD returns zero for success, or an error code for failure.

### Example

```
USE  IFLPORT
 CHARACTER(LEN=30) DIRECTORY
  ! variable DIRECTORY must be long enough to hold
  ! entire string
 INTEGER(4) ISTAT
    ISTAT = GETCWD (DIRECTORY)
    IF (ISTAT == 0) PRINT *, 'Current directory is ',
                                 DIRECTORY
    END
```

# GETDAT

*Returns the current date in integer form*

### Prototype

```
INTERFACE
    SUBROUTINE GETDAT(IYEAR,IMONTH,IDAY)
    INTEGER IYEAR,IMONTH,IDAY
    END SUBROUTINE GETDAT
END INTERFACE
```

IYEAR        an integer value

| IMONTH | an integer value |
| IDAY | an integer value |

### Description

This routine returns the current date in integer form. On Windows NT* systems, this function is thread-safe.

### Output

The current date is returned. IYEAR contains the current year, reckoned by the Julian calendar. IMONTH contains the numerical version of the month, where 1 corresponds to January, and 12 corresponds to December. IDAY returns the numerical day of the month.

# GETDRIVEDIRQQ

*Gets the complete path of the current working directory on a specified disk drive.*

### Prototype

```
USE IFLPORT
 or
INTERFACE
   ! Get the current directory for a given drive
  INTEGER(4) FUNCTION GETDRIVEDIRQQ(DRIVEDIR)
    CHARACTER(LEN=*) DRIVEDIR
  END FUNCTION
END INTERFACE
```

### Usage

```
result = GETDRIVEDIRQQ (DRIVEDIR)
```

DRIVEDIR          CHARACTER(LEN=*). For input, DRIVEDIR contains
                  the drive whose current working directory path is to be
                  returned. On output, DRIVEDIR contains the current
                  directory on that drive in the form d:\dir.

## Results

The result is INTEGER(4). The result is the length (in bytes) of the full
path on the specified drive. If the full path is longer than the size of
DRIVEDIR, zero is returned.

You can make sure you get information about the current drive, by putting
the symbolic constant FILE$CURDRIVE (defined in IFLPORT.F90) into
DRIVEDIR.

Since disk drives are identified by a single alphabetic character,
GETDRIVEDIRQQ examines only the first letter of DRIVEDIR. For
instance, if DRIVEDIR contains the path c:\program files,
GETDRIVEDIRQQ (DRIVEDIR) returns the current working directory on
drive C and disregards the rest of the path. Input is case-insensitive. The
length of the path returned depends on how deeply the directories are nested
on the drive specified in DRIVEDIR. If the full path is longer than the length
of DRIVEDIR, GETDRIVEDIRQQ returns only the portion of the path that
fits into DRIVEDIR. If you are likely to encounter a long path, allocate a
buffer of size MAXPATH (where MAXPATH is a PARAMETER constant
defined in IFLPORT.F90, and MAXPATH = 260).

## Example

```
!  Program to demonstrate GETDRIVEDIRQQ
USE IFLPORT
CHARACTER(MAXPATH) dir
INTEGER(4) length
!  Get current directory
dir = FILE$CURDRIVE
length = GETDRIVEDIRQQ(dir)
IF (length .GT. 0) THEN
  WRITE (*,*) 'Current directory is: '
  WRITE (*,*) dir
```

```
ELSE
  WRITE (*,*) 'Failed to get current directory'
END IF
END
```

# GETDRIVESIZEQQ

*Gets the total size of the specified*
DRIVE.

### Prototype

```
USE IFLPORT
or
INTERFACE GETDRIVESIZEQQ
  LOGICAL(4) FUNCTION GETDRIVESIZEQQI4  (DriveNm,
TotalNum, AvailableNum)
    CHARACTER(LEN=*) DriveNm
    INTEGER(4) TotalNum
    INTEGER(4) AvailableNum
  END FUNCTION
  LOGICAL(4) FUNCTION GETDRIVESIZEI8 (DriveNm,
TotalNum, AvailableNum)
     CHARACTER(LEN=*) DriveNm
    INTEGER(8) TotalNum
    INTEGER(8) AvailableNum
  END FUNCTION
END INTERFACE
```

### Description

Gets the total size of the specified DRIVE and space available on it.

### Usage

```
result = GETDRIVESIZEQQ (DRIVENm, TOTALNUM,
AVAILIABLENUM)
```

DriveNm          CHARACTER(LEN=*). String containing the letter of
                 the disk drive to get information about.

TotalNum         INTEGER(4) or INTEGER(8). TotalNum number of
                 bytes on the disk drive

AvailableNum     INTEGER(4) or INTEGER(8). Number of bytes of
                 AVAILable space on the disk drive.

## Results

The function returns LOGICAL(4). The result is .TRUE. if successful;
otherwise, .FALSE..

The data types specified for the TotalNum and AvailableNum
arguments must be the same.

Some disk drives have more bytes than will fit into a 32 bit integer. Using
an INTEGER(4) variable for these drives will return a negative size. To get
the actual size for thse large drives, TotalNum and AvailableNum must
be INTEGER(8) variables.

Because disk drives are identified by a single alphabetic character,
GETDRIVESIZEQQ examines only the first letter of DriveNm. The drive
letter can be uppercase or lowercase. You can use the constant
FILE$CURDRIVE (defined in IFLPORT.F90) to get the size of the current
DRIVE.

If GETDRIVESIZEQQ fails, use GETLASTERRORQQ to determine the
reason.

## Example

```
! Program to demonstrate GETDRIVESQQ and
GETDRIVESIZEQQ
USE IFLPORT
CHARACTER(26) drives
CHARACTER(1) adrive
LOGICAL(4) status
```

```fortran
INTEGER(4) total, avail
INTEGER(2) i
! Get the list of drives
drives = GETDRIVESQQ()
WRITE (*,'(A, A)') ' Drives available: ', drives
!
!Cycle through them for free space and write to
console
DO i = 1, 26
  adrive = drives(i:i)
  status = .FALSE.
  WRITE (*,'(A, A, A,)') ' Drive ', CHAR(i + 64), ':'
  IF (adrive .NE. ' ') THEN
      status = GETDRIVESIZEQQ(adrive, total, avail)
  END IF
  IF (status) THEN
      WRITE (*,*) avail, ' of ', total, ' bytes free.'
  ELSE
      WRITE (*,*) 'Not available'
  END IF
END DO
END
```

# GETDRIVESQQ

*Reports which drives are available to the system.*

### Prototype

```fortran
USE IFLPORT
```
or
```fortran
INTERFACE
```

```
      CHARACTER(26) FUNCTION GETDRIVESQQ()
      END FUNCTION
END INTERFACE
```

## Usage

```
result = GETDRIVESQQ ( )
```

## Results

The result is `CHARACTER(LEN=26)`. The returned string contains letters for drives that are available, and blanks for drives that are not available. For example, on a system with A, C, and D drives, the string 'A CD' is returned.

# GETENV

*Return the value of a system environment variable.*

## Prototype

```
INTERFACE
  SUBROUTINE GETENV (VAR, VALUE)
    CHARACTER(LEN=*) VAR, VALUE
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| VAR | must be `CHARACTER` type. Specifies the environment variable name. |
| VALUE | must be `CHARACTER` type. The `VALUE` is assigned the environment variable's value. `VALUE` must be declared large enough to hold the value. If the environment variable is not defined, `VALUE` is set to all blanks. |

### Description

Returns the value of a system environment variable.

### Class

Nonstandard subroutine.

### Example

The following code assigns VAL the value of the TERM environment variable.

```
CHARACTER(10) VAL
CALL GETENV("TERM", VAL)
```

# GETENVQQ

*Gets the value of a specified environment variable.*

### Prototype

```
USE IFLPORT
or
! ALTERNATIVE WAY OF GETTING ENVIRONMENT VARIABLE
VALUE
INTERFACE
  INTEGER(4) FUNCTION GETENVQQ(VARNAME,VALUE)
    CHARACTER(LEN=*) VARNAME,VALUE
  END FUNCTION GETENVQQ
END INTERFACE
```

### Description

Gets the value of a specified environment variable from the current environment.

## Usage

```
result = GETENVQQ (VARNAME, VALUE)
```

VARNAME      CHARACTER(LEN=*). The name of environment
             variable whose value you want to find.

VALUE        CHARACTER(LEN=*). Value of the specified
             environment variable, in uppercase.

## Results

The result type is INTEGER(4). The result is the number of characters
returned in VALUE. Zero is returned if the given variable is not defined.

GETENVQQ searches the list of environment variables for an entry
corresponding to VARNAME. Environment variables define the environment
in which a process executes. For example, the LIB environment variable
defines the default search path for libraries to be linked with a program.
Note that some environment variables may exist only on a per-process
basis, and may not be present at the command-line level.

## Example

```
!  Program to demonstrate GETENVQQ and SETENVQQ
USE IFLPORT
INTEGER(4) lenv, lval
CHARACTER(80) env, val, enval
WRITE (*,900) ' Enter environment variable name to
create, modify, or delete: '
lenv = GETSTRQQ(env)
IF (lenv .EQ. 0) STOP
WRITE (*,900) 'Value of variable (ENTER to delete): '
lval = GETSTRQQ(val)
IF (lval .EQ. 0) val = ' '
enval = env(1:lenv) // '=' // val(1:lval)
IF (SETENVQQ(enval)) THEN
  lval = GETENVQQ(env(1:lenv), val)
  IF (lval .EQ. 0) THEN
    WRITE (*,*) 'Can''t get environment variable'
  ELSE IF (lval .GT. LEN(val)) THEN
```

```
                 WRITE (*,*) 'Buffer too small'
           ELSE
               WRITE (*,*) env(:lenv), ': ', val(:lval)
               WRITE (*,*) 'Length: ', lval
           END IF
       ELSE
           WRITE (*,*) 'Can''t set environment variable'
       END IF
           FORMAT (A)
       END
```

# GETFILEINFOQQ

*Returns information about the specified
file.*

### Prototype

```
USE IFLPORT or
INTERFACE
   INTEGER(4) FUNCTION GETFILEINFOQQ(FILES, BUFFER,
                               DWHANDLE)
    CHARACTER(LEN=*) FILES
    TYPE FILE$INFO
    SEQUENCE
     INTEGER(4) CREATION   ! CREATION TIME (-1 ON FAT)
     INTEGER(4) LASTWRITE  ! LAST WRITE TO FILE
     INTEGER(4) LASTACCESS ! LAST ACCESS (-1 ON FAT)
     INTEGER(4) LENGTH     ! LENGTH OF FILE
     INTEGER(2) PERMIT     ! FILE ACCESS MODE
     CHARACTER(LEN=255) NAME  ! FILE NAME
    END TYPE
    TYPE(FILE$INFO) :: BUFFER
```

```
      INTEGER(4) DWHANDLE
  END FUNCTION GETFILEINFOQQ
END INTERFACE
```

### Description

Returns information about the specified file. Filenames can contain wildcards (* and ?).

### Usage

```
result = GETFILEINFOQQ (FILES,BUFFER, HANDLE)
```

FILES        `CHARACTER(LEN=*)`. Name or pattern of files that you are looking for information on. It can include a full path and can include wildcards (* and ?).

BUFFER      Derived type `FILE$INFO`. Information about a file that matches the search criteria is returned in this variable.

HANDLE     `INTEGER(4)`. Control mechanism. One of the following constants, defined in `IFLPORT.F90`:

           `FILE$FIRST`: First matching file found.

           `FILE$LAST`: Previous file was the last valid file.

           `FILE$ERROR`: No matching file found.

### Results

The result is `INTEGER(4)`, showing the nonblank length of the filename if a match was found, or 0 if no matching `FILES` were found.

To get information about one or more files, set `HANDLE` to `FILE$FIRST` and call `GETFILEINFOQQ`. This will return information about the first file which matches the name and return a `HANDLE`**.** If the program wants more files, it should call `GETFILEINFOQQ` with the `HANDLE`. `GETFILEINFOQQ` must be called with the `HANDLE` until `GETFILEINFOQQ` sets `HANDLE` to `FILE$LAST`, or system resources may be lost.

The derived-type element variables `FILE$INFO%CREATION`, `FILE$INFO%LASTWRITE`, and `FILE$INFO%LASTACCESS` contain packed date and time information that indicates when the file was created, last written to, and last accessed, respectively. To break the time and date

into component parts, call `UNPACKTIMEQQ`. `FILE$INFO%LENGTH` contains the length of the file in bytes. `FILE$INFO%PERMIT` contains a set of bit flags describing access information about the file as follows:

| Bit Flag | Corresponding File |
| --- | --- |
| `FILE$ARCHIVE` | Marked as having been copied to a backup device. |
| `FILE$DIR` | A subdirectory of the current directory. Each MS-DOS directory contains two special files, "." and "..". These are directory aliases created by MS-DOS for use in relative directory notation. The first refers to the current directory, and the second refers to the current directory's parent directory. |
| `FILE$HIDDEN` | Hidden. It does not appear in the directory list you request from the command line, the Microsoft visual development environment browser, or File Manager. |
| `FILE$READONLY` | Write-protected. You can read the file, but you cannot make changes to it. |
| `FILE$SYSTEM` | Used by the operating system. |
| `FILE$VOLUME` | A logical volume, or partition, on a physical disk drive. This type of file appears only in the root directory of a physical device. |

You can use the constant `FILE$NORMAL` to check that all bit flags are set to 0. If the derived-type element variable `FILE$INFO%PERMIT` is equal to `FILE$NORMAL`, the file has no special attributes. The variable `FILE$INFO%NAME` contains the short name of the file, not the full path of the file.

If an error occurs, call `GETLASTERRORQQ` to retrieve the error message, such as:

| | |
| --- | --- |
| `ERR$NOENT` | The file or path specified was not found. |
| `ERR$NOMEM` | Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly. |

### Example

```
USE IFLPORT
CALL SHOWPERMISSION( )
END
SUBROUTINE SHOWPERMISSION()
!  SUBROUTINE to demonstrate GETFILEINFOQQ
!
USE IFLPORT
!
CHARACTER(80) files
INTEGER(4) handle, length
CHARACTER(5) permit
TYPE (FILE$INFO) info
!
WRITE (*, 900) ' Enter wildcard of files to view: '
900   FORMAT (A)
length = GETSTRQQ(files)
handle = FILE$FIRST
DO WHILE (.TRUE.)
   length = GETFILEINFOQQ(files, info, handle)
   IF ((handle .EQ. FILE$LAST) .OR. &
       (handle .EQ. FILE$ERROR)) THEN
   SELECT CASE (GETLASTERRORQQ())
       CASE (ERR$NOMEM)
       WRITE (*,*) 'Out of memory'
       CASE (ERR$NOENT)
       EXIT
       CASE DEFAULT
    WRITE (*,*) 'Invalid file or path name'
       END SELECT
   END IF
     permit = ' '
     IF ((info%permit .AND. FILE$HIDDEN) .NE. 0) &
     permit(1:1) = 'H'
```

```
            IF ((info%permit .AND. FILE$SYSTEM) .NE. 0) &
        permit(2:2) = 'S'
            IF ((info%permit .AND. FILE$READONLY) .NE. 0) &
        permit(3:3) = 'R'
            IF ((info%permit .AND. FILE$ARCHIVE) .NE. 0) &
        permit(4:4) = 'A'
            IF ((info%permit .AND. FILE$DIR) .NE. 0) &
            permit(5:5) = 'D'
            WRITE (*, 9000) info%name, info%length, permit
9000  FORMAT (1X, A20, I9, ' ',A6)
END DO
END SUBROUTINE
```

# GETGID

*Gets the  group ID*

### Prototype

```
INTERFACE
  INTEGER FUNCTION GETGID ()
  END FUNCTION GETGID
END INTERFACE
```

### Description

This function returns an integer corresponding to the primary group of the user under whose identity this program is running.

On Win32* systems, this function returns the last subauthority of the security identifier for `TokenPrimaryGroup` for this process. This is unique on a local machine and unique within a domain for domain accounts.

**NOTE.** *You should be aware that on WIN32 systems, domain accounts and local accounts can overlap.*

### Output

An integer representing the group that the currently logged in user belongs to.

# GETLASTERROR

*Gets the last error set*

### Prototype

```
INTERFACE
  INTEGER FUNCTION GETLASTERROR ()
  END FUNCTION GETLAST
END INTERFACE
```

### Description

This function returns the integer corresponding to the last runtime error value that was set.

For example, if you use an `ERR=` `specifier` on an I/O statement, your program will not abort in the event of an error. `GETLASTERROR` provides a way to determine what the error condition was, with a better degree of certainty than just examining `errno`. Your application code may then take appropriate action based upon the error number.

**Output**

Last error number into an integer.

# GETLASTERRORQQ

*Returns the last error set by a run-time procedure.*

**Prototype**

```
USE IFLPORT
or
INTERFACE
  NTEGER(4) FUNCTION GETLASTERRORQQ()
  END FUNCTION
END INTERFACE
```

**Usage**

```
result = GETLASTERRORQQ ( )
```

**Results**

The result is `INTEGER(4)`, and shows the most recent error code generated by a run-time procedure.

Descriptions that return a logical or integer value sometimes also provide an error code that identifies the cause of errors. `GETLASTERRORQQ` retrieves the most recent error number, usually associated with errno. The error constants are in `IFLPORT.F90` The following table shows some of the portability library routines and the errors each routine produces:

| Runtime Routine | Runtime Routines Errors |
|---|---|
| RUNQQ | ERR$NOMEM, ERR$2BIG, ERR$INVAL, ERR$NOENT, ERR$NOEXEC |

| Runtime Routine | Runtime Routines Errors |
|---|---|
| SYSTEMQQ | ERR$NOMEM, ERR$2BIG, ERR$NOENT, ERR$NOEXEC |
| GETDRIVESIZEQQ | ERR$INVAL, ERR$NOENT |
| GETDRIVESQQ | no error |
| GETDRIVEDIRQQ | ERR$NOMEM, ERR$RANGE |
| CHANGEDRIVEQQ | ERR$INVAL, ERR$NOENT |
| CHANGEDIRQQ | ERR$NOMEM, ERR$NOENT |
| MAKEDIRQQ | ERR$NOMEM, ERR$ACCES, ERR$EXIST, ERR$NOENT |
| DELDIRQQ | ERR$NOMEM, ERR$ACCES, ERR$NOENT |
| FULLPATHQQ | ERR$NOMEM, ERR$INVAL |
| SPLITPATHQQ | ERR$NOMEM, ERR$INVAL |
| GETFILEINFOQQ | ERR$NOMEM, ERR$NOENT, ERR$INVAL |
| SETFILETIMEQQ | ERR$NOMEM, ERR$ACCES, ERR$INVAL, ERR$MFILE, ERR$NOENT |
| SETFILEACCESSQQ | ERR$NOMEM, ERR$INVAL, ERR$ACCES |
| DELFILESQQ | ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$INVAL |
| RENAMEFILEQQ | ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$XDEV |
| FINDFILEQQ | ERR$NOMEM, ERR$NOENT |
| PACKTIMEQQ | no error |
| UNPACKTIMEQQ | no error |
| COMMITQQ | ERR$BADF |
| GETCHARQQ | no error |
| PEEKCHARQQ | no error |
| GETSTRQQ | no error |
| GETLASTERRORQQ | no error |
| SETERRORMODEQQ | no error |
| GETENVQQ | ERR$NOMEM, ERR$NOENT |

| Runtime Routine | Runtime Routines Errors |
|---|---|
| SETENVQQ | ERR$NOMEM, ERR$INVAL |
| SLEEPQQ | no error |
| BEEPQQ | no error |
| SORTQQ | ERR$INVAL |
| BSEARCHQQ | ERR$INVAL |

# GETLOG

*Returns the user's login name*

## Prototype

```
INTERFACE
    SUBROUTINE GETLOG (NAME)
    CHARACTER(LEN=*) NAME
    END SUBROUTINE GETLOG
END INTERFACE
```

NAME                    the login name

## Description

This function allows your application to determine the login name of the person running the application. The login name must be less than 64 characters. If the login name is longer than 64 characters, it will be truncated. The actual parameter corresponding to the formal parameter NAME in the prototype should be long enough to hold the login name. If the supplied actual parameter is too short to hold the login name, the login name will be truncated.

**Output**

A character string in `NAME` corresponding to the login name of the person running the application. If the login name is shorter than the actual parameter corresponding to `NAME`, the login name is padded with blanks at the end, until it reaches the length of the actual parameter.

# GETPID

*Gets the process ID*

### Prototype

```
INTERFACE
  INTEGER FUNCTION GETPID ()
  END FUNCTION GETPID
END INTERFACE
```

### Description

This function returns the process `ID` of the current process.

### Output

A unique integer corresponding to the system process identifier for the current process.

# GETPOS

*Returns the current file position in bytes*
*from the beginning of the file*

### Prototype

```
INTERFACE
  INTEGER FUNCTION GETPOS (LUNIT)
```

```
      INTEGER LUNIT
    END FUNCTION GETPOS
END INTERFACE
```

LUNIT        INTEGER   Fortran logical unit number for a file. The value must be in the range 0 to 100 and must correspond to a connected file.

### Description

Allows you to determine the current file position.

### Output

An integer value representing the number of bytes from the beginning of the file. Equivalent to FTELL. Returns EINVAL in errno and a result of –1 for an error.

# GETSTATUSFPQQ

*Returns the floating-point processor status word.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  SUBROUTINE GETSTATUSFPQQ(STATUS)
    INTEGER(2)     STATUS
  END SUBROUTINE
END INTERFACE
```

### Usage

```
CALL GETSTATUSFPQQ (STATUS)
```

STATUS        INTEGER(2). Floating-point co-processor status word.

The floating-point status word (FPSW) shows whether various floating-point exception conditions have occurred. After an exception occurs, the runtime system does not reset flags before performing additional floating-point operations. A status flag with a value of one thus shows there has been at least one occurence of the corresponding exception. The following table lists the status flags and their values:

| Parameter Name | Hex Value | Description |
|---|---|---|
| FPSW$MSW_EM | Z'003F' | Status Mask (set all flags to 1) |
| FPSW$INVALID | Z'0001' | An invalid result occurred |
| FPSW$DENORMAL | Z'0002' | A denormal (very small number) occurred |
| FPSW$ZERODIVIDE | Z'0004' | A divide by zero occurred |
| FPSW$OVERFLOW | Z'0008' | An overflow occurred |
| FPSW$UNDERFLOW | Z'0010' | An underflow occurred |
| FPSW$INEXACT | Z'0020' | Inexact precision occurred |

You can use a bit-wise AND on the status word returned by GETSTATUSFPQQ to determine which floating-point exception has occurred.

An exception is disabled if its flag is set to 1 and enabled if its flag is cleared to 0. By default, the denormal, underflow and inexact precision exceptions are disabled, and the invalid, overflow and divide-by-zero exceptions are enabled. Exceptions can be enabled and disabled by clearing and setting the flags with SETCONTROLFPQQ. You can use GETCONTROLFPQQ to determine which exceptions are currently enabled and disabled.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, floating-point processes generate an appropriate special value (NaN or signed infinity), but the program continues. When printed, the runtime-system represents a NaN as ?????, positive infinity as ++++++, and negative infinity as -------. You can find out which exceptions (if any) occurred by calling GETSTATUSFPQQ.

If errors on floating-point exceptions are enabled (by clearing the flags to 0 with SETCONTROLFPQQ), an interrupt is generated when the exception occurs. By default, these interrupts cause run-time errors, but you can capture the interrupts with SIGNALQQ and branch to your own error-handling routines.

### Example

```
!  Program to demonstrate GETSTATUSFPQQ
 USE IFLPORT
 INTEGER(2) status
 CALL GETSTATUSFPQQ(status)
!  check for divide by zero
 IF (IAND(status, FPSW$ZERODIVIDE) .NE. 0) THEN
    WRITE (*,*) 'Divide by zero occurred.', &
             'Look for NaN or signed infinity in
                resultant data.'
 ELSE
     PRINT *,"Divide by zero flag was not set"
 END IF
END
```

# GETSTRQQ

*Reads a character string from the keyboard using buffered input.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  INTEGER(4) FUNCTION GETSTRQQ(BUFFER)
    CHARACTER(LEN=*) BUFFER
  END FUNCTION
```

```
END INTERFACE
```

## Usage

```
result = GETSTRQQ (BUFFER)
```

BUFFER          `CHARACTER(LEN=*)`. Character value returned from keyboard, padded on the right with blanks.

## Results

The result is the number of characters placed in `BUFFER`. The function continues, until the user presses Return or Enter.

## Example

```
!  Program to demonstrate GETSTRQQ
 USE IFLPORT
 INTEGER(4) length, result
 CHARACTER(80) prog, args
 WRITE (*, '(A,)') ' Enter program to run: '
 length = GETSTRQQ (prog)
 WRITE (*, '(A,)') ' Enter arguments: '
 length = GETSTRQQ (args)
 result = RUNQQ (prog, args)
 IF (result .EQ. -1) THEN
   WRITE (*,*) 'Couldn''t run program'
 ELSE
   WRITE (*, '(A, Z4, A)') 'Return code : ', result,
'h'
 END IF
 END
```

# GETTIM

*Returns the time*

### Prototype

```
INTERFACE
  SUBROUTINE GETTIM (HOUR, MINUTE, SECOND, HUNDREDTH)
  INTEGER HOUR, MINUTE, SECOND, HUNDREDTH
  END SUBROUTINE GETTIM
END INTERFACE
```

| | |
|---|---|
| HOUR | current system hour |
| MINUTE | current system minutes |
| SECOND | current system seconds |
| HUNDREDTH | current system hundredths of a second |

### Description

This function gets the current system time in hours, minutes, seconds, and hundredths of a second. The time units are returned as separate integers to the calling routine.

### Output

The time units are returned as separate integers to the calling routine.

# GETUID

*Gets the user ID of the calling process*

### Prototype

```
INTERFACE
```

```
    INTEGER FUNCTION GETUID()
    END FUNCTION GETUID
END INTERFACE
```

### Description

This function returns an integer corresponding to the user identity under which this program is running. This function returns the last subauthority of the security identifier for this process. This is unique on a local machine and unique within a domain for domain accounts.

---

**NOTE.** *You should be aware that on WIN32 systems, domain accounts and local accounts can overlap.*

---

### Output

The user ID of the calling process.

# GMTIME

*Converts the given elapsed time to the current system time*

### Prototype

```
INTERFACE
  SUBROUTINE GMTIME (STIME, DATEARRAY)
  INTEGER STIME, DATEARRAY(9)
  END SUBROUTINE GMTIME
END INTERFACE
```

STIME            STIME represents an elapsed time in seconds since midnight, January 1, 1970, in GMT. You can obtain this value from GETTIMEOFDAY, if you adjust the result of GETTIMEOFDAY for your local time zone.

DATEARRAY(1:9) contains the current date and system time, GMT.

DATEARRAY(1) seconds (0-59)

DATEARRAY(2) minutes (0-59)

DATEARRAY(3) hours (0-23)

DATEARRAY(4) day of month (1-31)

DATEARRAY(5) month (0-11)

DATEARRAY(6) year in century (0-99)

DATEARRAY(7) day of week (0-6, 0 is Sunday)

DATEARRAY(8) day of year (0-365)

DATEARRAY(9) daylight savings (1, if in effect; else 0)

### Description

This function converts a given elapsed time in seconds into the current system date, GMT.

**NOTE.** *This function may give problems with the year 2000. Use DATE_AND_TIME instead.*

### Output

The current date and system time for Greenwich Mean Time, in DATEARRAY.

# GRAN

*Generate Gaussian normal random*
*numbers*

### Prototype

```
INTERFACE
  REAL (4) FUNCTION GRAN ()
  END FUNCTION GRAN
END INTERFACE
```

### Class

Specific elemental nonstandard function.

### Description

Generate Gaussian normal random numbers.

# HOSTNAM

*Retrieves the current host computer*
*name.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  INTEGER(4) FUNCTION HOSTNAM(NAME)
    CHARACTER(LEN=*),INTENT(OUT):: NAME
  END FUNCTION
END INTERFACE
```

### Description

Retrieves the current host computer name. This function is exactly the same as HOSTNM.

### Usage

```
result = HOSTNAM (NAME)
```

NAME            CHARACTER(LEN=*). The name of the current computer host is returned in NAME. The buffer provided should be at least as long as MAX_COMPUTERNAME_LENGTH, which is defined in the IFLPORT module.

### Results

The result is zero if successful. If NAME is not long enough to contain all of the host name, the function truncates the host name and returns -1.

### Example

```
USE IFLPORT
CHARACTER(LEN=15) HOSTNAME
INTEGER(4)  ISTATUS
ISTAT = HOSTNAM (HOSTNAME)
PRINT *, HOSTNAME, ' ISTATUS = ', ISTATUS
END
```

# HOSTNM

*Gets the current host name*

### Prototype

```
INTERFACE
  SUBROUTINE HOSTNM( NAME )
  CHARACTER(LEM=*) NAME
```

```
      END SUBROUTINE HOSTNM
END INTERFACE
```

NAME                A CHARACTER variable or array element large enough
                    to hold the name of the current host computer. On
                    Win32 systems, the maximum host name length is 15.

### Description

This function retrieves the current host computer name.

### Output

An ASCII character string specifying the name of the host computer where
your application is executing.

# IARGC

*Returns the number of arguments on the
command line*

### Prototype

```
INTERFACE
  INTEGER FUNCTION IARGC ()
  END FUNCTION IARC
END INTERFACE
```

### Description

This function returns the number of arguments on the command line, not
including the command itself.

### Output

Number of arguments into an integer. If no arguments are passed to the
program, IARGC returns zero.  Otherwise IARGC returns a count of the
arguments that follow the program name on the command line.

**Example**

The statement `PRINT *,IARGC()` prints a count of the arguments passed to the program.

# IDATE

*Returns the current system date*

**Prototype**

```
INTERFACE
    SUBROUTINE IDATE(MON, DAY, YEAR)
    INTEGER MON, DAY, YEAR
    END SUBROUTINE IDATE
END INTERFACE
```

MON             an integer value

DAY             an integer value

YEAR            an integer value

**Description**

This routine returns the current system month, day, and year.

**Example**

The statement `CALL IDATE (MON,DAY,YR)` sets `MON` to the month number, `DAY` to the day of the month, and `YR` to the two-digit year representation (for example, 69 for the year 1969).

**NOTE.** *This subroutine is not year-2000 compliant. Use DATE_AND_TIME or IDATE4 instead.*

**Output**

MON                 is returned with current system month.

DAY                 is returned with current system day.

YEAR                is returned with current system year as on offset from
                    1900.

# IDATE4

*Returns the current system date*

### Prototype

```
INTERFACE
    SUBROUTINE IDATE4(DATEARRAY)
    INTEGER DATEARRAY(3)
    END SUBROUTINE IDATE4
END INTERFACE
```

DATEARRAY       an integer array.

### Description

This routine returns the current system month, day, and year.

This function is year-2000 compliant.

### Output

DATEARRAY(1)        is returned with current system day.

DATEARRAY(2)        is returned with current system month.

DATEARRAY(3)        is returned with current system year as an offset
                    from 1900, if the year is less than 2000. For years
                    greater than or equal to 2000, this element simply
                    returns the integer year, such as 2003.

# IEEE_FLAGS

*Sets IEEE flags*

### Prototype

```
INTERFACE
  INTEGER FUNCTION IEEE_FLAGS (ACTION, MODE, IN, OUT)
  CHARACTER(LEN=*) ACTION, MODE, IN, OUT
  END FUNCTION IEEE_FLAGS
END INTERFACE
```

| | |
|---|---|
| ACTION | **is one of**: 'GET', 'SET', 'CLEAR' |
| MODE | **is one of**: 'direction', 'precision', 'exception' |
| IN | **is one of**: 'inexact', 'division', 'underflow', 'overflow', 'invalid', 'all', 'common', 'nearest-to-zero', 'negative', 'positive', 'extended', 'double', 'single' |
| OUT | if MODE is 'direction', OUT **is one of**: 'nearest-to-zero', 'negative', 'positive' |
| | if MODE is 'precision', OUT **is one of**: 'extended', 'double', 'single' |
| | if MODE is 'exception', OUT **is one of**: 'inexact', 'division', 'underflow', 'overflow', 'invalid' |

### Description

IEEE_FLAGS is an elemental function that sets IEEE flags for GET, SET, and CLEAR procedures.

### Output

Returns 0 if executes successfully, 1 otherwise.

# IEEE_HANDLER

*Establishes a handler for IEEE exceptions*

### Prototype

```
INTERFACE
  INTEGER FUNCTION IEEE_HANDLER (ARG_ACTION, &
    ARG_EXCEPTION, HANDLER)
  CHARACTER(LEN=*) ARG_ACTION, ARG_EXCEPTION
   INTERFACE
    SUBROUTINE HANDLER (SIGNO, SIGINFO)
       INTEGER(4), INTENT(IN)::SIGNO, SIGINFO
    END SUBROUTINE
   END INTERFACE
  END FUNCTION IEEE_HANDLER
END INTERFACE
```

### Description

`IEEE_HANDLER` calls `HANDLER` subroutine to establish a handler for IEEE exceptions.

### Output

Returns 0 if executes successfully, 1 otherwise.

# IERRNO

*Returns the last error code generated*

### Prototype

```
INTERFACE
```

```
        INTEGER FUNCTION IERRNO()
        END FUNCTION IERRNO
END INTERFACE
```

**Description**

This function returns the number of the last detected error from any module that returns error codes.

**Output**

The last error code generated.

# IFL_RUNTIME_INIT

*Initializes the Fortran runtime system, complete with command line arguments*

**Prototype**

```
INTERFACE
 SUBROUTINE IFL_RUNTIME_INIT ( NCMDARGS, ARGS )
   INTEGER(4) NCMDARGS
   INTEGER(1), POINTER :: ARGS
END INTERFACE
```

| | |
|---|---|
| NCMDARGS | The number of arguments on the command line for the application. |
| ARGS | A pointer to an array of character strings, where each string makes up a command line argument |

**Description**

This routine initializes the Fortran runtime system, including supplying command line arguments. Normally this is not required. However, if your main program is written in C, or your main program is a WINMAIN in C or

C++ for Windows\*, there may be circumstances where you wish to set the command line arguments for the Fortran portion of the program, and perform other initialization. For IA-32 Windows applications, you would call this routine from C as:

```
extern void IFL_RUNTIME_INIT ( int *argc, char
                                  *argv[]);

  int main(int *argc, char *argv[] )
  {
    IFL_RUNTIME_INIT( argc, argv);
  }
```

# IFLOATI

*Converts an INTEGER(2) to a REAL  type*

### Prototype

```
INTERFACE
  REAL(4) FUNCTION IFLOATI (INPUT)
  INTEGER(2), INTENT(IN)::INPUT
  END FUNCTION IFLOATI
END INTERFACE
```

INPUT              a scalar INTEGER (KIND=2) value

### Description

IFLOATI is an elemental function that converts a scalar integer (KIND=2) type to REAL(4).

### Output

The integer value converted to REAL.

# INMAX

*Returns the maximum positive integer*

### Prototype

```
INTERFACE
  INTEGER FUNCTION INMAX (I)
  INTEGER I
  END FUNCTION INMAX
END INTERFACE
```

I                an INTEGER(4) value

### Description

This function returns the maximum positive value for an INTEGER(4).

### Output

The maximum 4-byte-signed integer.

# INTC

*ConvertsINTEGER(4) to INTEGER(2)*

### Prototype

```
INTERFACE
  INTEGER(2) FUNCTION INTC (IN)
  INTEGER IN
  END FUNCTION INTC
END INTERFACE
```

IN                any INTEGER(4) value or expression

### Description

Converts an `INTEGER(4)` value or expression to an `INTEGER(2)` value.

### Output

The value of `IN` converted to a type `INTEGER(2)`. Overflow is ignored.

# IOMSG

### Description

Print the text for an I/O message.

### Class

Nonstandard subroutine.

# IRAND

*Generates pseudorandom numbers.*

```
INTERFACE
  INTEGER(4) FUNCTION IRAND()
  INTEGER(4) ISEED
  END FUNCTION IRAND
END INTERFACE
```

### Description

Generates pseudorandom numbers.

### Class

Elemental nonstandard function.

### Result Type and Type Parameter

`INTEGER(4)` type.

### Result Value

`IRAND` generates numbers in the range 0 through $2^{31}$.

### Example

```
INTEGER(4) rn
rn = IRAND()
```

> **NOTE.** *For details about restarting the pseudorandom number generator used by* `IRAND` *and* `RAND`*, see the "s*RAND*" section.*

# IRANDM

*A synonym for* `IRAND`.

### Description

This function is a synonym for IRAND. It takes the same parameters and produces the same result.

# IRANGET

*Gets a random number*

### Prototype

```
INTERFACE
  SUBROUTINE IRANGET (S)
  INTEGER S
  END SUBROUTINE IRANGET
END INTERFACE
```

S                           an integer expression

### Description

Returns the next in a series of pseudo-random numbers.

### Output

An integer pseudo random number.

# IRANSET

*Sets the seed for a sequence of
pseudo-random numbers*

### Prototype

```
INTERFACE
  SUBROUTINE IRANSET (ISEED)
  INTEGER ISEED
  END SUBROUTINE IRANSET
END INTERFACE
```

ISEED            the new seed for a sequence of pseudo-random numbers

### Description

Sets the seed for a sequence of random numbers.

### Output

Changes the internal seed. Not thead-safe.

## ISATTY

*Returns true if the specified unit number
is a terminal*

### Prototype

```
INTERFACE
  LOGICAL FUNCTION ISATTY (LUN)
  INTEGER LUN
  END FUNCTION ISATTY
END INTERFACE
```

LUN            an integer expression corresponding to a Fortran logical
unit number in the range of 0 to 100

### Description

This function returns true if the specified unit number is a terminal. The unit
must be connected.

LUN must be an integer expression. If LUN is out of range, zero is returned.
LUN must map to a connected Fortran logical unit at the time of the call. If
LUN corresponds to a unit that is not connected, zero is returned.

**Output**

.TRUE. for a logical unit connected to a terminal device. .FALSE. otherwise.

# ITIME

*Returns the time*

### Prototype

```
INTERFACE
    SUBROUTINE ITIME(TIME_ARRAY)
    INTEGER TIME_ARRAY(3)
    END FUNCTION ITIME
END INTERFACE
```

TIME_ARRAY      an integer array

### Description

This function returns the time in numeric form in a 3-element array.

### Output

TIME_ARRAY[1] contains the hour.

TIME_ARRAY[2] contains the minutes.

TIME_ARRAY[3] contains the seconds.

# JABS

*Returns the absolute value*

### Prototype

```
INTERFACE
```

```
             INTEGER FUNCTION JABS (X)
             INTEGER X
             END FUNCTION JABS
         END INTERFACE
```

# JDATE

*Returns the date in ASCII*

### Prototype

```
INTERFACE
   SUBROUTINE JDATE(CURRENTDATE)
   CHARACTER(LEN=8) CURRENTDATE
   END SUBROUTINE JDATE
END INTERFACE
```

### Description

This function returns an 8-character ASCII string with the Julian date, (day of the year).

### Output

An 8-character ASCII string with the Julian date in the form `yyddd`.

**NOTE.** *Use of this function is discouraged due to possible problems with the change to the year 2000. Use* `DATEANDTIME`, *a standard function instead*.

# JDATE4

*Returns the date in ASCII*

### Prototype

```
INTERFACE
  SUBROUTINE JDATE4(CURRENTDATE)
  CHARACTER(LEN=10) CURRENTDATE
  END SUBROUTINE JDATE4
END INTERFACE
```

### Description

This function returns an 10-character ASCII string with the Julian date, (day of the year).

### Output

An 10-character ASCII string with the Julian date in the form yyyyddd.

# KILL

*Stops an active process*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION KILL (PID, SIGNUM)
  INTEGER(4) PID, SIGNUM
  END FUNCTION
END INTERFACE
```

PID                       process ID

SIGNUM              signal value

## Description

This function kills the process associated with the specified PID. This function requires that the program executing this function have TERMINATE_PROCESS access to the process being killed.

## Output

If successful, zero.

# LCWRQQ

*Sets the value of the floating-point processor control word.*

## Prototype

```
USE IFLPORT
or
INTERFACE
 SUBROUTINE LCWRQQ(CONTROL)
    INTEGER(2) CONTROL
 END SUBROUTINE
END INTERFACE
```

## Usage

```
CALL LCWRQQ (CONTROL)
```

CONTROL          INTEGER(2). Floating-point processor control word.

LCWRQQ performs the same function as SETCONTROLFPQQ and is provided for compatibility.

## Example

```
USE IFLPORT
```

```
INTEGER(2) control
CALL SCWRQQ(control) ! get control word
!  Set control word to make processor round up
control = control .AND. (.NOT. FPCW$MCW_RC)  ! Clear
                                        ! control
word with inverse
                                            ! of
rounding control mask
control = control .OR. FPCW$UP  ! Set control word
                                ! to round up
CALL LCWRQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

# LTIME

*Returns the current time and system
time*

### Prototype

```
INTERFACE
    SUBROUTINE LTIME(TIME,DATEARRAY)
    INTEGER TIME, DATEARRAY(9)
    END SUBROUTINE
END INTERFACE
```

TIME                 elapsed time in seconds since 00:00:00 Greenwich
Mean Time, January 1, 1970

### Description

This routine returns the current time since 0:00:00 January 1, 1970 in the
TIME variable, and the current system time in the components of the array.

### Output

Returns the components of the local time in a nine element array:

| | |
|---|---|
| DATEARRAY(1) | Seconds (0-59) |
| DATEARRAY(2) | Minutes (0-59) |
| DATEARRAY(3) | Hours (0-23) |
| DATEARRAY(4) | Day of month (1-31) |
| DATEARRAY(5) | Month (0-11) |
| DATEARRAY(6) | Year in century (0-99) |
| DATEARRAY(7) | Day of week (0-6, where 0 is Sunday) |
| DATEARRAY(8) | Day of year (1-365) |
| DATEARRAY(9) | 1 if daylight savings time in effect. |

---

**NOTE.** *This function is not year 2000 compliant, use DATE_AND_TIME instead.*

---

# MAKEDIRQQ

*Creates a new directory with a specified name.*

### Prototype

```
USE IFLPORT
or
 INTERFACE
    LOGICAL(4) FUNCTION MAKEDIRQQ(DIRNAME)
      CHARACTER(*)    DIRNAME
    END FUNCTION
 END INTERFACE
```

## Usage

```
result = MAKEDIRQQ (DIRNAME)
```

DIRNAME       `CHARACTER(LEN=*)`. Name and path of the directory you want created.

## Results

`MAKEDIRQQ` returns `.TRUE.` if successful; otherwise, `.FALSE.`.

`MAKEDIRQQ` can create only one directory at a time. You cannot create a new directory and a subdirectory below it in a single command. `MAKEDIRQQ` does not translate path delimiters. You can use either slash (/) or backslash (\) as valid delimiters.

If an error occurs, you should call `GETLASTERRORQQ` to determine the problem. Possible errors include:

`ERR$ACCES`       Permission denied. The file's (or directory's) permission setting does not allow the specified access.

`ERR$EXIST`       The directory already exists.

`ERR$NOENT`       The file or path specified was not found.

## Example

```
USE IFLPORT
LOGICAL(4) result
result = MAKEDIRQQ('mynewdir')
IF (result) THEN
   WRITE (*,*) 'New subdirectory successfully
created'
ELSE
   WRITE (*,*) 'Failed to create subdirectory'
END IF
END
```

# MALLOC

*Allocates a block of memory*

### Prototype

```
INTERFACE
    INTEGER FUNCTION MALLOC(SIZE)
    INTEGER SIZE
    END FUNCTION MALLOC
END INTERFACE
```

SIZE         number of bytes of memory to be allocated

### Description

Allocates a block of memory.

> **NOTE.** *Do not use it in conjunction with* ALLOCATE *and* DEALLOCATE.*Use with Cray pointers only.*

### Output

The starting address of the allocated memory.

# MATHERRQQ

*Handles run-time math errors.*

### Prototype

```
INTERFACE
    SUBROUTINE MATHERRQQ( NAME,NLEN,INFO,RETCODE)
```

```
CHARACTER(LEN=*) NAME
INTEGER(2)  :: NLEN, RETCODE
STRUCTURE /MTH$E_INFO/
 INTEGER*4 ERRCODE    ! INPUT : One of the MTH$
                      ! values above
 INTEGER*4 FTYPE      ! INPUT : One of the TY$
                      ! values above
 UNION
 MAP
  REAL*4 R4ARG1       ! INPUT : FIrst argument
  CHARACTER*12 R4FILL1
  REAL*4 R4ARG2        ! INPUT : Second argument
                       !(if any)
  CHARACTER*12 R4FILL2
  REAL*4 R4RES         ! OUTPUT : Desired result
  CHARACTER*12 R4FILL3
 END MAP
 MAP
   REAL*8 R8ARG1       ! INPUT : FIrst argument
   CHARACTER*8 R8FILL1
   REAL*8 R8ARG2        ! INPUT : Second argument
                        !(if any)
   CHARACTER*8 R8FILL2
   REAL*8 R8RES         ! OUTPUT : Desired result
   CHARACTER*8 R8FILL3
 END MAP
 MAP
   COMPLEX*8 C8ARG1    ! INPUT : FIrst argument
   CHARACTER*8 C8FILL1
   COMPLEX*8 C8ARG2    ! INPUT : Second argument
                       ! (if any)
   CHARACTER*8 C8FILL2
   COMPLEX*8 C8RES     ! OUTPUT : Desired result
   CHARACTER*8 C8FILL3
 END MAP
```

```
        MAP
          COMPLEX*16 C16ARG1   ! INPUT : FIrst argument
          COMPLEX*16 C16ARG2   ! INPUT : Second argument
                               ! (if any)
          COMPLEX*16 C16RES    ! OUTPUT : Desired result
        END MAP
      END UNION
    END STRUCTURE
    RECORD /MTH$E_INFO/ info
  END SUBROUTINE
END INTERFACE
```

**NOTE.** *Due to the fact that you cannot use old VAX style structures in Fortran 95* MODULE*s with Intel Fortran, there is no interface for this function in* IFLPORT.F90

## Usage

```
CALL MATHERRQQ (NAME,NLEN,INFO,RETCODE)
```

NAME            CHARACTER(LEN=*). On return, this variable contains the name of the function causing the error. The parameter NAME is a typeless version of the function called. For example, if an error occurs in a SIN function, the name will be returned as SIN for real arguments and CSIN for complex arguments even though the function may have actually been called with an alternate name such as DSIN or CDSIN, or with SIN and complex arguments.

NLEN            INTEGER(2). Number of characters returned in NAME.

INFO            UNION containing data about the error. The MTH$E_INFO union is defined above.

RETCODE         INTEGER(2). Return code passed back to the run-time library. The value of RETCODE should be set by the user's MATHERRQQ routine to indicate whether the error

was resolved. Set this value to 0 to indicate that the error was not resolved and that the program should fail with a run-time error. Set it to any nonzero value to indicate that the error was resolved and the program should continue.

The `ERRCODE` element in the `MTH$E_INFO` structure specifies the type of math error that occurred, and can have one of the following values:

| Vlaue | Meaning |
|---|---|
| `MTH$E_DOMAIN` | Argument domain error |
| `MTH$E_OVERFLOW` | Overflow range error |
| `MTH$E_PLOSS` | Partial loss of significance |
| `MTH$E_SINGULARITY` | Argument singularity |
| `MTH$E_TLOSS` | Total loss of significance |
| `MTH$E_UNDERFLOW` | Underflow range error |

The `FTYPE` element of the `INFO` structure identifies the data type of the math function as `TY$REAL4`, `TY$REAL8`, `TY$CMPLX4`, or `TY$CMPLX8`. Internally, `REAL(4)` and `COMPLEX(4)` arguments are converted to `REAL(8)` and `COMPLEX(8)`. In general, a `MATHERRQQ` function should test the `FTYPE` value and take separate action for `TY$REAL8` or `TY$CMPLX8` using the appropriate mapped values. If you want to resolve the error, set the `R8RES` or `C8RES` field to an appropriate value such as 0.0. You can do calculations within the `MATHERRQQ` function using the approprate `ARG1` and `ARG2` fields, but avoid doing any calculations that would cause an error resulting in another call to `MATHERRQQ`.

**NOTE.**  *You cannot use MATHERRQQ in DLLs or in a program that links with a DLL.*

# NARGS

*Returns the number of arguments on the command line*

### Prototype

```
INTERFACE
  INTEGER FUNCTION NARGS ()
  END FUNCTION NARGS
END INTERFACE
```

### Description

This function returns the number of arguments on the command line, not including the invoking command itself.

### Output

An integer value, zero or positive, indicating the number of arguments on the command line invoking your program.

# NUMARG

*Returns the number of arguments on the command line*

### Prototype

```
INTERFACE
  INTEGER FUNCTION NUMARG ()
  END FUNCTION NUMARG
END INTERFACE
```

### Description

This function returns the number of arguments on the command line, not including the invoking command itself.

### Output

An integer value, zero or positive, indicating the number of arguments on the command line invoking your program.

# PACKTIMEQQ

*Packs time and date values.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  SUBROUTINE
    PACKTIMEQQ(TIMEDATE,IYR,IMON,IDAY,IHR,IMIN,ISEC)
    INTEGER(4) TIMEDATE
    INTEGER(2) IYR, IMON, IDAY, IHR, IMIN, ISEC
  END SUBROUTINE
 END INTERFACE
```

### Usage

```
CALL PACKTIMEQQ
 (TIMEDATE,IYR,IMON,IDAY,IHR,IMIN,ISEC)
```

TIMEDATE       INTEGER(4). Packed time and date information.

IYR            INTEGER(2). Year (xxxx AD).

IMON           INTEGER(2). Month (1 - 12).

IDAY           INTEGER(2). Day (1 - 31)

IHR            INTEGER(2). Hour (0 - 23)

IMIN            INTEGER(2). Minute (0 - 59)

ISEC            INTEGER(2). Second (0 - 59)

The packed time is the number of seconds since 00:00:00 Greenwich mean time, January 1, 1970. You can numerically compare packed time items. You can use PACKTIMEQQ to work with relative date and time values. Use UNPACKTIMEQQ to unpack time information. SETFILETIMEQQ uses packed time.

### Example

```
USE IFLPORT
 INTEGER(2) year, month,day,hour,minute,second,hund
 INTEGER(4) timedate
 INTEGER(4) y4,m4,d4,h4,s4,hu4
 CALL GETDAT (y4, m4, d4)
 year = y4
 month = m4
 day = d4
 CALL GETTIM (h4, m4, s4, hu4)
 hour = h4
 minute = m4
 second = s4
 hund = hu4
CALL PACKTIMEQQ (timedate, year, month, day, hour,
& minute, second)
END
```

# PEEKCHARQQ

*Checks the keystroke buffer for a recent console keystroke.*

### Prototype

```
USE IFLPORT
or
INTERFACE
! TEST FOR CONSOLE INPUT
  LOGICAL(4) FUNCTION PEEKCHARQQ()
  END FUNCTION PEEKCHARQQ
END INTERFACE
```

### Description

Checks the keystroke buffer for a recent console keystroke and returns `.TRUE.` if there is a character in the buffer or `.FALSE.` if there is not.

### Usage

```
result = PEEKCHARQQ ( )
```

### Results

The result type is LOGICAL(4). The result is .TRUE. if there is a character waiting in the keyboard buffer; otherwise, .FALSE..

To find out the value of the key in the buffer, call `GETHARQQ`. If there is no character waiting in the buffer when you call `GETCHARQQ`, `GETCHARQQ` waits until there is a character in the buffer. If you call `PEEKCHARQQ` first, you prevent `GETCHARQQ` from halting your process while it waits for a keystroke. If there is a keystroke, `GETCHARQQ` returns it and resets `PEEKCHARQQ` to `.FALSE.`.

### Example

```
USE IFLPORT
```

```
LOGICAL(4) pressed / .FALSE. /
DO WHILE (.NOT. pressed)
  WRITE(*,*) ' Press any key'
  pressed = PEEKCHARQQ ( )
END DO
END
```

# PERROR

*Sends a message to standard error*

### Prototype

```
INTERFACE
  SUBROUTINE PERROR (STRING)
  CHARACTER(LEN=*) STRING
  END SUBROUTINE PERROR
END INTERFACE
```

STRING           message to precede the standard error message

### Description

Sends a message to the standard error stream, preceded by the specified
STRING.

# POPCNT

*Counts the number of 1-bits in the given value*

### Prototype

```
INTERFACE
  INTEGER FUNCTION POPCNT (VALUE)
  INTEGER VALUE
  END FUNCTION POPCNT
END INTERFACE
```

VALUE                a positive integer value

### Description

This function counts the number of 1-bits in the given value.

### Output

Number of 1-bits.

# POPPAR

*Population parity*

### Prototype

```
INTERFACE
  INTEGER FUNCTION POPPAR (P)
  TYPE P
  END FUNCTION POPPAR
END INTERFACE
```

TYPE          may have the following types: `BYTE`, `INTEGER`, `INTEGER(2)`, `INTEGER(4)`, `LOGICAL(1)`, `LOGICAL(2)`, `LOGICAL(4)`, `REAL(4)`, `POINTER`

P             any scalar value up to a maximum of 32 bits in length

### Description

This function returns `0` if the number of bits in the argument is even, `1` if the number is odd.

## PUTC

*Writes a character to standard output.*

### Prototype

```
INTERFACE
    INTEGER FUNCTION PUTC(CH)
    CHARACTER(LEN=1) CH
    END FUNCTION PUTC
END INTERFACE
```

CH             a character variable

### Description

Writes a character to the standard output device. Intel Fortran assumes that external unit 6 is connected to the standard output device (`stdout`), which is where your output will be sent. Typically `stdout` is your terminal screen. If unit 6 is connected to some other device, this routine will still send output to the standard output device.

### Output

A zero if successful; otherwise, an error code.

# QSORT

*Sorts an array*

### Prototype

```
INTERFACE
   SUBROUTINE QSORT(ARRAY, LEN, ISIZE, COMP)
     TYPE ARRAY(LEN)
   INTEGER LEN,ISIZE
   INTERFACE
     INTEGER(2) FUNCTION COMP(P1, P2)
     TYPE P1, P2
     END FUNCTION COMP
     END INTERFACE
     END SUBROUTINE QSORT
   END INTERFACE
```

| | |
|---|---|
| `ARRAY` | is a one-dimensional array of any of the following |
| `TYPE` | is any intrinsic or derived type |
| `LEN` | is the number of elements in the array |
| `ISIZE` | is the size in bytes of a single element of the array |
| `COMP` | is a comparison function that you must supply that returns |

      `<0`, if `P1 .LT. P2`
      `=0`, if `P1 = P2`
      `>0`, if `P1 .GT. P2`

### Description

Sorts the given array using the given comparison function.

> **NOTE.** QSORT *can provide unpredictable results for multi-dimensional arrays. It assumes C language element order and one-dimensional arrays.*

### Output

QSORT returns your array sorted in place in ascending order.

# RAISEQQ

*Sends a signal to the executing program.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  INTEGER(4) FUNCTION RAISEQQ(SIGNUMBER)
      INTEGER(4)       SIGNUMBER
  END FUNCTION
END INTERFACE
```

### Usage

```
result = RAISEQQ (SIGNUMBER)
```

SIGNUMBER      the number of the signal to raise. One of the following constants (defined in IFLPORT.F90 )

SIG$ABORT      Abnormal termination

SIG$FPE        Floating-point error

SIG$ILL             Illegal instruction

SIG$INT             CTRL+C signal

SIG$SEGV            Illegal storage access

SIG$TERM            Termination request

If you do not install a signal handler (with SIGNALQQ, for example), when a signal occurs the system by default terminates the program.

### Results

The result is zero if successful; otherwise, nonzero.

If a signal-handling routine for SIGNUMBER has been installed by a prior call to SIGNALQQ, RAISEQQ causes that routine to be executed. If no handler routine has been installed, the system terminates the program (the default action).

# RAN

*Generates a random number between 0 and 1.*

### Prototype

```
INTERFACE
    REAL(4) FUNCTION RAN(ISEED)
    END FUNCTION RAN
END INTERFACE
```

ISEED           must be an INTEGER(4) variable or array element. RAN stores a number in ISEED to be used by the next call to RAN. ISEED should initially be set to an odd number, preferably very large; see the Example.

### Description

Generates random numbers between 0 and 1 from an integer seed ISEED. It updates ISEED with the new value of the internal seed. At the first call, it is recommended to initialize ISEED to a large, odd value.

### Class

Elemental nonstandard function.

### Example

```
INTEGER(4) iseed
REAL(4) rnd
iseed = 425001
rnd = RAN(iseed)
```

> **NOTE.** *To ensure different random values for each run of a program,* ISEED *should be set to a different value each time the program is run. One way to implement this would be to have the user enter the seed at the start of the program. Another way would be to compute a value from the current year, day, and month (returned by* IDATE) *and the number of seconds since midnight.*

# RAND

*Generates a random number in the range of 0.0 to 1.0.*

### Prototype

```
INTERFACE
   REAL(4) FUNCTION RAND(ISEED)
   INTEGER(4) ISEED
   END FUNCTION RAND
```

```
END INTERFACE
```

### Description

Generate a random number uniformly distributed in the range of 0.0 to 1.0. At the first call, it is recommended to set ISEED to a large, odd value to initialize the internal seed. ISEED is not updated. For subsequent calls, ISEED should be set to zero to get the next random number in the sequence. If ISEED is set to one, the internal seed can also be set by

```
CALL SEED(ISEED).
```

### Class

Elemental nonstandard function.

### Result Type and Parameter Type

REAL(4) type.

### Output

The output depends on the parameter value as follows:

*   If the input parameter is equal to zero, RAND returns the next number in the pseudorandom sequence.
*   If the input parameter is equal to one, RAND restarts the pseudorandom number sequence and returns the first number of the sequence.
*   If the input parameter is greater than one, the value is used as the seed for a new pesudorandom sequence, and RAND returns the first number in that sequence.

### Example

```
INTERFACE
   REAL(4) FUNCTION RAND(ISEED)
   INTEGER(4) ISEED
END INTERFACE
INTEGER(4) ISEED
REAL(4) rv
rv = RAND(ISEED)
```

```
END
```

**NOTE.** *For details about restarting the pseudorandom number generator used by* IRAND *and* RAND, *see the "*SRAND*" section.*

# RANDOM

*Random number generator*

### Prototype

```
INTERFACE
  SUBROUTINE RANDOM (R)
  REAL (4) R
  END SUBROUTINE RANDOM
END INTERFACE
```

R               a REAL(4) variable or array element that will contain the random number on return

### Description

RANDOM generates a pseudo-random number, based upon the value of the seed set by the RANSET function. Portability functions RANDOM, RAND, and DRAND all generate the same results. A given seed will always generate the same sequence of pseudo-random numbers. RANDOM is an implementation of the algorithm described in *Random Number Generators*: *Good ones are hard to find*, by Park, S.K.,and Miller, K.W., in Comm ACM, Oct. 1988, 1192-1201. This is also described in section 7 of Numerical Recipes. This routine is provided for compatibility with legacy FORTRAN programs. You should use the RANDOM_NUMBER and RANDOM_SEED intrinsic functions that are standard  Fortran  when possible.

### Output

Random real number.

# RANDU

*Generates a pseudorandom number* in
the range 0.0 to 1.0.

### Prototype

```
INTERFACE
  REAL(4) FUNCTION RANDU (IL, I2, X)
  END FUNCTION RANDU
END INTERFACE
```

IL, I2          must be `INTEGER(2)` variables or array elements that contain the `SEED` for computing the random number. These values are updated during the computation so that they contain the updated seed.

X          A `REAL(4)` variable or array element where the computed random number is returned.

### Result

The result is returned in `X`, which must be of type `REAL(4)`. The result value is a pseudorandom number in the range 0.0 to 1.0.

The algorithm for computing the random number value is based on the values for `IL` and `I2`.

If `IL=0` and `I2=0`, the generator base is set as follows:

$$X(N + 1 = 2^{16} + 3)$$

Otherwise, it is set as follows:

$$X(N + 1 = (2^{16} + 3) * X(N) \bmod 2^{32})$$

The generator base `X(N + 1)` is stored in `IL, I2`.

The result is          `X(N + 1)`

scaled to a real value    `Y(N + 1),`

for              $0.0 <= Y(N + 1) < 1.$

### Example

```
REAL X
INTEGER(2) I, J
...
CALL RANDU (I, J, X)
```

If I and J are values 4 and 6, X stores the value 5.4932479E-04.

# RANF

*Generates a random number betweem*
*0. and `RAND_MAX`.*

### Prototype

```
INTERFACE
  REAL(4) FUNCTION RANF ()
  END FUNCTION RANF
END INTERFACE
```

### Description

`RANF` returns a single-precision pseudo-random number between 0.0 and `RAND_MAX` as defined in the C library, normally 0x7FFF $2^{15}$-1. The initial seed is set by

```
CALL SRAND(ISEED)
```

### Output

A random number of the type `REAL(4)`. `ISEED` is of `INTEGER(4)`.

# RANGET

*Returns the current seed*

### Prototype

```
INTERFACE
  SUBROUTINE RANGET (S)
  INTEGER S
  END SUBROUTINE RANGET
END INTERFACE
```

### Description

RANGET returns the current seed used for a sequence of psuedo-random numbers.

### Output

The internal seed. This routine is not thread-safe.

# RANSET

*Sets the seed for the random number generator*

### Prototype

```
INTERFACE
  SUBROUTINE RANSET (ISEED)
  REAL ISEED
  END SUBROUTINE RANSET
END INTERFACE
```

### Description

Sets the seed for a sequence of pseudo-random numbers.

### Output

A changed seed.

# RENAME

*Renames a file*

### Prototype

```
INTERFACE
    FUNCTION RENAME(FROM,TO)
    CHARACTER(LEN=*) FROM,TO
    END FUNCTION RENAME
END INTERFACE
```

FROM            a character path name of the origin file

TO              a character path name that specifies the name and
                location where you wish to place the FROM file

### Description

This routine renames a file. Either FROM or TO may include a fully qualified or relative path name. RENAME accepts either forward or backward slashes as directory separators, and converts them to the form appropriate for the host operating system. On NT, you may include drive letters in the path also. Drive letters are not accepted on UNIX systems.

The FROM file must exist when rename is called, but you do not have to have it connected to a logical unit.

---

**NOTE.** *The* FROM *and* TO *paths for the files must be on the same physical device. You cannot use this routine to move a file from one device to another.*

---

This routine is thread-safe, and locks the associated stream before I/O is performed.

### Output

A zero status is returned for success, non-zero for failure.

# RENAMEFILEQQ

*Renames a file.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  LOGICAL(4) FUNCTION RENAMEFILEQQ(OLDNAME,NEWNAME)
   CHARACTER(LEN=*) OLDNAME,NEWNAME
  END FUNCTION RENAMEFILEQQ
END INTERFACE
```

### Usage

```
result = RENAMEFILEQQ (OLDNAME,NEWNAME)
```

OLDNAME          CHARACTER(LEN=*). File to be renamed.

NEWNAME          CHARACTER(LEN=*). New name of the file to be renamed.

### Results

The result is .TRUE. if successful; otherwise, .FALSE..

You can use RENAMEFILEQQ to move a file from one directory to another on the same drive by giving a different path in the NEWNAME parameter.

If the function fails, you should call GETLASTERRORQQ to determine the reason. One of the following errors can be returned:

ERR$ACCES     Permission denied. The file's permission setting does not allow the specified access.

ERR$EXIST     The file already exists.

ERR$NOENT     File or path specified by OLDNAME not found.

ERR$XDEV      Attempt to move a file to a different device.

### Example

```
USE iflport
 INTEGER(4) len
 CHARACTER(80) oldname, newname
 LOGICAL(4) result
 WRITE(*,'(A)') ' Enter old name: '
 len = GETSTRQQ(oldname)
 WRITE(*,'(A)') ' Enter new name: '
 len = GETSTRQQ(newname)
 result = RENAMEFILEQQ(oldname, newname)
 END
```

# RINDEX

*Locates the index of the last occurrence of a substring within a string*

### Prototype

INTERFACE

```
        INTEGER FUNCTION RINDEX (S1, S2, S1LEN, S2LEN)
        CHARACTER(LEN=*) S1, S2
        INTEGER S1LEN, S2LEN
        END FUNCTION RINDEX
END INTERFACE
```

| | |
|---|---|
| S1 | original string to search |
| S2 | string to search for |
| S1LEN | length of S1 string |
| S2LEN | length of S2 string |

### Description

This function locates the index of the last occurrence of a substring within a string.

### Output

Starting position of the final occurrence of S2 in S1.

# RUNQQ

*Executes another program and waits for it to complete.*

### Prototype

```
USE IFLPORT
or
INTEGER(2) FUNCTION RUNQQ(PROGNAME,COMMANDLINE)
    CHARACTER(LEN=*) PROGNAME, COMMANDLINE
END FUNCTION
```

### Usage

```
result = RUNQQ (FILENAME, COMMANDLINE)
```

FILENAME          Input. `CHARACTER(LEN=*)`. Filename of a program to be executed.

COMMANDLINE       Input. `CHARACTER(LEN=*)`. Command-line arguments passed to the program to be executed.

### Results

The result type is `INTEGER(2)`. If the program executed with `RUNQQ` terminates normally, the exit code of that program is returned to the program that launched it. If the program fails, -1 is returned.

The `RUNQQ` function executes a new process for the operating system using the same path, environment, and resources as the process that launched it. The launching process is suspended until execution of the launched process is complete.

### Example

```
USE IFLPORT
INTEGER(2) result
result = RUNQQ('dir', '/Os')
END
```

# SCWRQQ

*Returns the floating-point processor
control word.*

### Prototype

```
USE IFLPORT
or
        INTERFACE
        SUBROUTINE SCWRQQ(CONTROL)
            INTEGER(2)      CONTROL
        END SUBROUTINE
```

```
        END INTERFACE
```

### Usage

`CALL SCWRQQ (CONTROL)`

CONTROL             Output. `INTEGER(2)`. Floating-point processor control word.

`SCRWQQ` performs the same function as `GETCONTROLFPQQ`, and is provided for compatibility.

# SCANENV

*Scans the environment for environment variable.*

### Prototype

```
INTERFACE
  SUBROUTINE SCANENV (ENVNAME, ENVTEXT, ENVVALUE)
  CHARACTER(LEN=*) INTENT(IN)::ENVNAME
  CHARACTER(LEN=*) INTENT(OUT)::ENVTEXT, ENVVALUE
  END SUBROUTINE SCANENV
END INTERFACE
```

ENVNAME             a `CHARACTER` variable containing the name of an environment variable you need to find the value for.

ENVTEXT             set to the full text of the environment variable found, or to a " " if nothing is found.

ENVVALUE             set to the value associated with the environment found or " " if nothing is found.

### Description

Scans the environment for an environment variable that matches `ENVNAME` and returns the value or string it is set to.

**Output**

The text string or the value of the environment variable.

# SEED

*Sets the starting point for the random
number generator*

### Prototype

```
INTERFACE
   SUBROUTINE SEED (ISEED)
   INTEGER ISEED
   END SUBROUTINE SEED
END INTERFACE
```

### Description

Sets the internal seed for a sequence of pseudo-random numbers.

### Output

A changed seed. This routine is not thread-safe.

# SECNDS

*Returns the number of seconds since
last call*

### Prototype

```
INTERFACE
   REAL FUNCTION SECNDS(TIME)
   REAL TIME
```

```
    END FUNCTION SECNDS
END INTERFACE
```

TIME                0.0 to start the clock, or the last value returned from
                    SECNDS

## Description

This function returns the elapsed time in seconds since the last call to
SECNDS, or the number of seconds since midnight if the parameter is 0.0.

## EXAMPLE

```
        PROGRAM TIMEIT
        REAL STARTTIME,STOPTIME
        STARTTIME= SECNDS(0.0)
        DO 10 I = 1,100000
        I = I +1
10      CONTINUE
        STOPTIME= SECNDS(STARTTIME)
        PRINT *, 'Elapsed time was: ',STOPTIME
        END
```

We provide this routine for compatibility with other FORTRAN compilers.
You can time a section of your code's execution, or time your whole
program. You can also time sections of your program and add the times
together. This routine is primarily useful in benchmarking, or as a rough
profiling guide of where your application spends most of its execution time.

To start the timing clock, call SECNDS with 0.0, and save the result in a
local variable. To get the elapsed time since the last call to SECNDS, pass
the local variable to SECNDS on the next call.

This routine is thread-safe.

## Output

The elapsed time in seconds since midnight, or since midnight minus the
value of the supplied floating point argument. If you want more accurate
timing, see the subroutine DCLOCK.

# SETCONTROLFPQQ

*Sets the value of the floating-point
processor control word.*

## Prototype

```
USE IFLPORT
or
INTERFACE
  SUBROUTINE SETCONTROLFPQQ(CONTROL)
    INTEGER(2) CONTROL
  END SUBROUTINE
END INTERFACE
```

## Usage

```
CALL SETCONTROLFPQQ (CONTROLWORD)
```

CONTROLWORD    INTEGER(2). Floating-point processor control word.

The floating-point control word allows you to specify how various
exception conditions are handled by the floating-point math coprocessor.
You can also set the floating-point precision, and specify the floating-point
rounding mechanism used.

The IFLPORT.F90 module file contains constants defined for the control
word as follows:

| Parameter Name | Hex Value | Description |
| --- | --- | --- |
| FPCW$MCW_IC | Z'1000' | **Infinity control mask** |
| FPCW$AFFINE | Z'1000' | Affine infinity |
| FPCW$PROJECTIVE | Z'0000' | Projective infinity |
| FPCW$MCW_PC | Z'0300' | **Precision control mask** |
| FPCW$64 | Z'0300' | 64-bit precision |
| FPCW$53 | Z'0200' | 53-bit precision |

| Parameter Name | Hex Value | Description |
|---|---|---|
| FPCW$24 | Z'0000' | 24-bit precision |
| FPCW$MCW_RC | Z'0C00' | **Rounding control mask** |
| FPCW$CHOP | Z'0C00' | Truncate |
| FPCW$UP | Z'0800' | Round up |
| FPCW$DOWN | Z'0400' | Round down |
| FPCW$NEAR | Z'0000' | Round to nearest |
| FPCW$MCW_EM | Z'003F' | **Exception mask** |
| FPCW$INVALID | Z'0001' | Allow invalid numbers |
| FPCW$DENORMAL | Z'0002' | Allow denormals (very small numbers) |
| FPCW$ZERODIVIDE | Z'0004' | Allow divide by zero |
| FPCW$OVERFLOW | Z'0008' | Allow overflow |
| FPCW$UNDERFLOW | Z'0010' | Allow underflow |
| FPCW$INEXACT | Z'0020' | Allow inexact precision |

The defaults for the floating-point control word are 53-bit precision, round to nearest, and the denormal, underflow and inexact precision exceptions disabled. An exception is disabled if its flag is set to 1 and enabled if its flag is cleared to 0.

Setting the floating-point precision and rounding mechanism can be useful if you are reusing old code that is sensitive to the floating-point precision standard used and you want to get the same results as on the old machine.

You can use GETCONTROLFPQQ to retrieve the current control word and SETCONTROLFPQQ to change the control word. If you need to change the control word, always use SETCONTROLFPQQ to make sure that special routines handling floating-point stack exceptions and abnormal propagation work correctly.

### Example

```
USE IFLPORT
INTEGER(2) status, control, controlo
```

```
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
 !      Save old control word controlo = control
 !      Clear all flags control = control .AND. Z'0000'
 !      Set new control to round up
 control = control .OR. FPCW$UP
 CALL SETCONTROLFPQQ(control)
 CALL GETCONTROLFPQQ(control)
 WRITE (*, 9000) 'Control word: ', control
 9000 FORMAT (1X, A, Z4)
 END
```

# SETDAT

*Sets the current system date in years, months, and day*

### Prototype

```
INTERFACE
  SUBROUTINE FUNCTION SETDAT (YEAR, MONTH, DAY)
  INTEGER YEAR, MONTH, DAY
  END FUNCTION SETDAT
END INTERFACE
```

| | |
|---|---|
| YEAR | 4-digit integer |
| MONTH | between 1-12 |
| DAY | between 1-31 |

### Description

This subroutine sets the current system date in years, months, and day. If the values are not valid, the date is not set.

**Output**

Changed date on the system executing the program.

# SETENVQQ

*Sets the value of an existing environment variable.*

### Prototype

```
USE IFLPORT
or
LOGICAL(4) FUNCTION SETENVQQ(INPUT_STRING)
    CHARACTER(LEN=*) INPUT_STRING
END FUNCTION SETENVQQ
```

### Description

Sets the value of an existing environment variable, or adds and sets a new environment variable.

### Usage

```
result = SETENVQQ (INPUT_STRING)
```

INPUT_STRING  CHARACTER(LEN=*). String containing both the name and the value of the variable to be added or modified. Must be in the form: VARNAME = VALUE, where VARNAME is the name of an environment variable and value is the value being assigned to it.

### Results

The result is .TRUE. if successful; otherwise, .FALSE..

Environment variables define characteristics of the environment in which a program executes. For example, the LIB environment variable defines the default search path for libraries to be linked with a program.

SETENVQQ deletes any terminating blanks in INPUT_STRING. Although the equal sign (=) is an illegal character within an environment value, you can use it to terminate VALUE so that trailing blanks are preserved. For example, the string PATH= = sets *value* to ".

You can use SETENVQQ to remove an existing variable by giving a variable name followed by an equal sign with no value. For example, LIB= removes the variable LIB from the list of environment variables. If you specify a value for a variable that already exists, its value is changed. If the variable does not exist, it is created.

SETENVQQ affects only the environment that is local to the current process. You cannot use it to modify the command-level environment. When the current process terminates, the environment reverts to the level of the parent process. In most cases, this is the operating system level. However, you can pass the environment modified by SETENVQQ to any child process created by RUNQQ, PXFFORK, or other means of creating a chid process.. These child processes get new variables and/or values added by SETENVQQ.

### Example

```
USE IFLPORT
! Note, compile this example with /nbs
LOGICAL(4) success
success = SETENVQQ("PATH=c:\users\mjsmith\bin")
success = &
SETENVQQ("LIB=c:\program
files\intel\compiler4.5\lib")
PRINT *, SUCCESS
END
```

# SETERRORMODEQQ

*Sets the prompt mode for critical errors.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  SUBROUTINE SETERRORMODEQQ(PROMPT)
    LOGICAL(4) PROMPT
  END SUBROUTINE
END INTERFACE
```

### Description

Sets the prompt mode for critical errors that by default generate system prompts.

### Usage

```
CALL SETERRORMODEQQ (PROMPT)
```

PROMPT            LOGICAL(4). PROMPT determines whether a prompt is displayed when a critical error occurs.

Certain I/O errors cause the system to display an error prompt. For example, attempting to write to a disk drive with the drive door open generates an "Abort, Retry, Ignore" message. When  your program begins execution, system error prompting is enabled by default. You can  enable system error prompts by calling SETERRORMODEQQ with PROMPT set to ERR$HARDPROMPT (defined in IFLPORT.F90 ).

If you disable prompting, serious I/O errors that would normally result in a prompt are silent.

Errors in I/O statements such as `OPEN`, `READ`, and `WRITE` fail immediately instead of being interrupted with prompts. This gives you more direct control of what happens when there is an error. You can use the `ERR=` facility of many I/O statements to give a label to branch to for error handling .

You can turn off prompt mode by setting `PROMPT` to `.FALSE.` or to the constant `ERR$HARDFAIL` (defined in `IFLPORT.F90`). You should be aware that `SETERRORMODEQQ` affects only errors that generate a system prompt. It does not affect other I/O errors, such as writing to a nonexistent file or attempting to open a nonexistent file with `STATUS='OLD'`.

### Example 1

```
!PROGRAM 1
 !  DRIVE B door open
 ! Note: you should compile these examples with /nbs
 OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
 !  Generates a system prompt error here and waits for
 ! the user to respond to the prompt before continuing
 100   WRITE(*,*) ' Continuing'
 END
```

### Example 2

```
 ! PROGRAM 2
 !  DRIVE B door open
 ! Note: you should compile these examples with /nbs
  USE IFLPORT
  CALL SETERRORMODEQQ(.FALSE.)
  OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
 !  Causes the statement at label 100 to execute
 !  without system prompt
 100   WRITE(*,*) ' Drive B: not available, opening
& &alternative drive.'
  OPEN (10, FILE = 'C:\NOFILE.DAT')
  END
```

# SETFILETIMEQQ

*Sets the modification time for a specified file.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  LOGICAL(4) FUNCTION SETFILETIMEQQ(NAME, TIMEDATE)
    CHARACTER(LEN=*) NAME
    INTEGER(4) TIMEDATE
   END FUNCTION
 END INTERFACE
```

### Usage

```
result = SETFILETIMEQQ (NAME, TIMEDATE)
```

NAME            CHARACTER(LEN=*). Name of a file.

TIMEDATE        INTEGER(4). Time and date information, as packed by PACKTIMEQQ.

### Results

The result is .TRUE. if successful; otherwise, .FALSE..

The modification time is the time the file was last modified.The process that calls SETFILETIMEQQ must have write access to the file; otherwise, you cannot change the time. If you set TIMEDATE to FILE$CURTIME (defined in IFLPORT.F90), SETFILETIMEQQ sets the modification time to the current system time.

If the function fails, call GETLASTERRORQQ to determine the reason, which can be one of the following:

| Error | Meaning |
|---|---|
| ERR$ACCES | Permission denied. The file's (or directory's) permission setting does not allow the specified access. |
| ERR$INVAL | Invalid argument; TIMEDATE argument is invalid. |
| ERR$MFILE | Too many open files (the file must be opened to change its modification time). |
| ERR$NOENT | File or path not found. |
| ERR$NOMEM | Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly. |

### Example

```
USE IFLPORT
INTEGER(2) day, month, year
INTEGER(2) hour, minute, second, hund
INTEGER(4) timedate
INTEGER(4) hr1, mn1,se1,hu1,ye1,mo1,da1
LOGICAL(4) result
CALL GETDAT(ye1, mo1, da1)
year = ye1
month = mo1
day = da1
CALL GETTIM(hr1, mn1, se1, hu1)
hour = hr1
minute = mn1
second = se1
hund = hu1
CALL PACKTIMEQQ (timedate, year, month, day,     &
                 hour, minute, second)
result = SETFILETIMEQQ('myfile.dat', timedate)
```

```
PRINT *, RESULT
END
```

# SETTIM

*Sets the current system date in years,*
*months, and days*

### Prototype

```
INTERFACE
   SUBROUTINE FUNCTION SETTIM (HOUR, MINUTE, SECOND, &
    HUNDREDTH)
   INTEGER HOUR, MINUTE, SECOND, HUNDREDTH
   END FUNCTION SETTIM
END INTERFACE
```

| | |
|---|---|
| HOUR | between 0 - 23 |
| MINUTE | between 0 - 59 |
| SECOND | between 0 - 59 |
| HUNDREDTH | between 0 - 99 |

### Description

This subroutine sets the current system time in hours, minutes, seconds and hundredths of a second. If the values are not valid, the time is not set.

### Output

Changed current time on the system executing the program.

# SHIFTL

*Shifts a value to the left by a specified number of bit positions*

## Prototype

```
INTERFACE
    INTEGER(4) FUNCTION SHIFTL(IVALUE,ISHIFTCOUNT)
    INTEGER(4) IVALUE,ISHIFTCOUNT
    END INTEGER FUNCTION SHIFTL
END INTERFACE
```

IVALUE            an integer value

ISHIFTCOUNT       the number of bit positions to shift. Must be positive
                  INTEGER(4) expression

## Description

This is an arithmetic shift. A shift count greater than the size in bits of the input value returns a result of zero.

This routine is thread-safe.

## Output

The input value is shifted left by ISHIFTCOUNT bit positions.

# SHIFTR

*Shifts a value to the right by a specified
number of bit positions*

### Prototype

```
INTERFACE
    INTEGER(4) FUNCTION SHIFTR(IVALUE,ISHIFTCOUNT)
    INTEGER(4) IVALUE,ISHIFTCOUNT
    END INTEGER FUNCTION SHIFTR
END INTERFACE
```

IVALUE                an integer value

ISHIFTCOUNT       the number of bit positions to shift. Must be positive
                     `INTEGER(4)` expression

### Description

This is an arithmetic shift. A shift count greater than the size in bits of the
input value returns a result of zero.

This routine is thread-safe.

### Output

The input value is shifted right by `ISHIFTCOUNT` bit positions.

# SIGNALQQ

*Registers the function to be called if an
interrupt signal occurs.*

### Prototype

```
USE IFLPORT
```

```
or
INTERFACE
          INTEGER*4 FUNCTION SIGNALQQ(SIGNAL, HANDLER)
!MS$ATTRIBUTES c,alias:'_signal' :: SIGNALQQ
             INTEGER*4 SIGNAL
!MS$ ATTRIBUTES VALUE ::   SIGNAL
             INTEGER(4) HANDLER
             EXTERNAL HANDLER
          END FUNCTION
       END INTERFACE
```

## Usage

`result = SIGNALQQ (SIGNAL,HANDLER)`

SIGNAL          `INTEGER(2)`. Interrupt type. You should specify one of the following constants, defined in `IFLPORT.F90`:

| | |
|---|---|
| SIG$ABORT | Abnormal termination |
| SIG$FPE | Floating-point error |
| SIG$ILL | Illegal instruction |
| SIG$INT | CTRL+C SIGNAL |
| SIG$SEGV | Illegal storage access |
| SIG$TERM | Termination request |

HANDLER         `CHARACTER(LEN=*)`. You should give a name of function to be executed on interrupt. The function must exist.

## Results

The result is a positive integer if successful; otherwise, -1 (`SIG$ERR`).

`SIGNALQQ` establishes the function `HANDLER` as the handler for a signal of the type specified by `SIGNAL`. If you do not establish a handler, the program terminates when an interrupt signal occurs.

The argument `HANDLER` is the name of a function and must be declared with either the `EXTERNAL` or `IMPLICIT` statements, or have an explicit interface. A function described in an `INTERFACE` block is `EXTERNAL` by default, and does not need to be declared `EXTERNAL`.

When an interrupt occurs, except a `SIG$FPE` interrupt, the `SIGNAL` argument SIG$INT is passed to `HANDLER`, and then `HANDLER` is executed.

When a SIG$FPE occurs, the function `HANDLER` is passed two arguments: SIG$FPE and the floating-point error code (for example, `FPE$ZERODIVIDE` or `FPE$OVERFLOW`) which identifies the type of floating-point exception that occurred. The floating-point error codes begin with the prefix `FPE$` and are defined in `IFLPORT.F90`.

If `HANDLER` returns, the calling process resumes execution immediately after the point where it received the interrupt signal. This is true regardless of the type of signal or operating mode.

Because signal-handler routines are normally called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. Therefore, in a signal handler routine, you should  not call heap routines or any routine that uses the heap routines (for example, I/O routines, `ALLOCATE`, and `DEALLOCATE`).

To test your signal handler routine you can generate interrupt signals by calling `RAISEQQ`, which causes your program either to branch to the signal handlers set with `SIGNALQQ`, or to perform the system default behavior if `SIGNALQQ` has set no signal handler.

The example below demonstrates a signal handler for `SIG$ABORT`.

### Example

```
!  This program shows a signal handler for
!  SIG$ABORT
 USE IFLPORT
 INTERFACE
   FUNCTION h_abort (signum)
     INTEGER(4) h_abort
     INTEGER(2) signum
   END FUNCTION
```

```
END INTERFACE
INTEGER(2) i2ret
INTEGER(4) i4ret
i4ret = SIGNALQQ(SIG$ABORT, h_abort)
WRITE(*,*) 'Set signal handler. Return = ', i4ret
i2ret = RAISEQQ(SIG$ABORT)
WRITE(*,*) 'Raised signal. Return = ', i2ret
END
!       Signal handler routine
INTEGER(4) FUNCTION h_abort (signum)
  INTEGER(2) signum
  WRITE(*,*) 'In signal handler for SIG$ABORT'
  WRITE(*,*) 'signum = ', signum
  h_abort = 1
END
```

# SLEEP

*Suspends execution of a process for a
specified interval*

### Prototype

```
INTERFACE
  SUBROUTINE SLEEP (TIME)
  INTEGER TIME
  END SUBROUTINE SLEEP
END INTERFACE
```

TIME                length of time, in seconds, to suspend the calling
                    process.

### Description

This function suspends the execution of a process for a specified interval.

### Output

None.

# SLEEPQQ

*Delays execution of the program for a specified duration.*

### Prototype

```
USE IFLPORT
or
INTERFACE
   SUBROUTINE SLEEPQQ( DURATION)
      INTEGER(4) DURATION
   END SUBROUTINE
END INTERFACE
```

### Usage

```
CALL SLEEPQQ (DURATION)
```

DURATION          INTEGER(4). Number of milliseconds the program is to sleep (delay program execution).

### Example

```
USE IFLPORT
INTEGER(4) delay, freq, duration
delay    = 2000
freq     = 4000
duration = 1000
CALL SLEEPQQ(delay)
```

```
CALL BEEPQQ(freq, duration)
END
```

# SPLITPATHQQ

*Breaks a file path or directory path into its components.*

### Prototype

```
USE IFLPORT
or
INTERFACE
   INTEGER(4) FUNCTION SPLITPATHQQ(PATH, DRIVE, DIR,
               NAME, EXT)
     CHARACTER(LEN=*) NAME, EXT, PATH, DRIVE, DIR
    END FUNCTION
 END INTERFACE
```

### Usage

```
result = SPLITPATHQQ (PATH,DRIVE,DIR,NAME,EXT)
```

| | |
|---|---|
| PATH | CHARACTER(LEN=*). Path that you want to break into components. Forward slashes (/), backslashes (\), or both can be present in PATH. |
| DRIVE | CHARACTER(LEN=*). Drive letter designation followed by a colon. |
| DIR | CHARACTER(LEN=*). Path of directories, including the trailing slash. |
| NAME | CHARACTER(LEN=*). Name of file or, if no file is specified in PATH, name of the lowest directory. If a filename, does not include an extension. |
| EXT | CHARACTER(LEN=*). Filename extension, if any, including the leading period (.). |

### Results

The result is the length of DIR.

PATH can be a complete or partial file specification.

MAXPATH is a symbolic constant defined in module IFLPORT.F90 as 260.

### Example

```
USE IFLPORT
CHARACTER(MAXPATH) buf
CHARACTER(3)        drive
CHARACTER(256)      dir
CHARACTER(256)      name
CHARACTER(256)      ext
CHARACTER(256)      file
INTEGER(4)          length
 ! Note, this example should be compiled with /nbs
buf = 'b:\fortran\test\runtime\tsplit.for'
length = SPLITPATHQQ(buf, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext
file = 'partial.f90'
length = SPLITPATHQQ(file, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext
END
```

# SRAND

*Restart the pseudorandom number generator used by IRAND and RAND.*

### Prototype

```
INTERFACE
  SUBROUTINE SRAND (ISEED)
```

```
    INTEGER ISEED
  END SUBROUTINE SRAND
END INTERFACE
```

ISEED           must be of INTEGER(4) type. The same value for
                ISEED generates the same sequence of random
                numbers.  To vary the sequence, call SRAND with a
                different ISEED value each time the program is
                executed.  The default for ISEED is 1.

### Description

Restart the pseudorandom number generator used by IRAND and RAND.

### Class

Specific nonstandard subroutine.

### Example

```
CALL SRAND(5041)
```

# SSWRQQ

*Returns the floating-point processor
status word.*

### Prototype

```
USE IFLPORT
or
INTERFACE
  SUBROUTINE SSWRQQ(STATUS)
    INTEGER(2) STATUS
  END SUBROUTINE
END INTERFACE
```

### Usage

```
CALL SSWRQQ (STATUS)
```

STATUS            INTEGER(2). Floating-point co-processor status word.

SSWRQQ performs the same function as GETSTATUSFPQQ and is provided for compatibility.

### Example

```
    USE IFLPORT
    INTEGER(2) status
    CALL SSWRQQ (status)
    PRINT 10,STATUS
10 FORMAT("FP Status word was ",Z)
    END
```

# STAT

*Gets system information on a given file*

### Prototype

```
INTERFACE
    INTEGER FUNCTION STAT(NAME,STATARRAY)
    CHARACTER(LEN=20) NAME
    INTEGER STATARRAY(13)
    END SUBROUTINE  STAT
END INTERFACE
```

NAME              a CHARACTER variable that specifies a pathname for the file that you want to check.

STATARRAY         an integer array where system information about your file can be placed

### Description

The filename must be currently connected to a logical unit, and must already exist when stat is called.

This routine is thread-safe, and locks the associated stream before information is collected.

> **NOTE.** *On NT,* stat *and* lstat *are equivalent. On UNIX, if the file denoted by* NAME *is a* link, *lstat provides information on the link, while* stat *provides information on the file at the destination of the link.*

### Output

Errno set on failure, otherwise, STATARRAY filled in with information about the file. The contents of the STATARRAY on return are system dependent.

| | | |
|---|---|---|
| stat(1) | Device that specifies the inode or handle | (always 0 on NT) |
| stat(2) | Inode or handle number | (always 0 on NT) |
| stat(3) | Protection level | |
| stat(4) | Number of hard links to file | (always 1 on NT) |
| stat(5) | User ID of owner | (always 1 on NT) |
| stat(6) | Group ID of owner | (always 1 on NT) |
| stat(7) | Device type, if this inode is a device | (always 0 on NT) |
| stat(8) | Total size of file | |
| stat(9) | File last access time | (for NT, only if file system non-FAT) |
| stat(10) | File last modify time | |
| stat(11) | File last status change time | (for NT, same as stat(10) |
| stat(12) | Optimal blocksize for file system I/O ops | (always 1 on NT) |

| | | |
|---|---|---|
| `stat(13)` | Actual number of blocks allocated | (Unix only, not present on NT) |

# SYSTEM

*Sends a command to the shell for execution*

### Prototype

```
INTERFACE
  INTEGER FUNCTION SYSTEM (COMMANDA)
  CHARACTER(LEN=*) COMMANDA
  END FUNCTION SYSTEM
END INTERFACE
```

COMMANDA        command to execute

### Description

This function sends a command to the shell for execution as if it were typed on the command line.

### Output

Exit status of the shell command.

# SYSTEMQQ

*Executes a system command.*

### Prototype

```
USE IFLPORT
```

or

```
INTERFACE
  LOGICAL(4) FUNCTION SYSTEMQQ(COMMANDA)
    CHARACTER(LEN=*) COMMANDA
    !MS$ATTRIBUTES ALIAS:'_SYSTEM' :: SYSTEMQQ
  END FUNCTION SYSTEMQQ
END INTERFACE
```

## Description

Executes a system command by passing a command string to the operating system's command interpreter.

## Usage

`result = SYSTEMQQ (COMMANDA)`

COMMANDA      `CHARACTER(LEN=*)`. Text of the command line to be passed to the operating system.

## Results

The result is `.TRUE.` if successful; otherwise, `.FALSE.`.

The `SYSTEMQQ` function lets you pass command shell commands as well as programs. `SYSTEMQQ` refers to the `COMSPEC` and `PATH` environment variables that locate the command interpreter file (usually named `COMMAND.COM`).

On Windows NT systems, the calling process waits until the command terminates. On Windows 95 and Windows 98 systems, the calling process does not currently wait in all cases; however, this may change in future implementations. To insure compatibility and consistent behavior, an image can be invoked directly by using the WIN32 API `CreateProcess ( )` in your Fortran code.

If the function faiils, you can call `GETLASTERRORQQ` to determine the reason. One of the following errors will be returned:

ERR$2BIG      The argument list exceeds 128 bytes, or the space required for the environment formation exceeds 32K.

ERR$NOINT      The command interpreter cannot be found.

| | |
|---|---|
| ERR$NOEXEC | The command interpreter file has an invalid format and is not executable. |
| ERR$NOMEM | Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly. |

The command line character limit for the SYSTEMQQ function is the same limit that your command shell accepts.

### Example

```
USE IFLPORT
! Note : compile this example with /nbs
LOGICAL(4) RESULT
RESULT = SYSTEMQQ('dir "c:\program files"')
PRINT *,'******'
PRINT *, 'Result returned from SYSTEMQQ was ',RESULT
END
```

# TIME

*Returns the current time*

### Prototype

```
INTERFACE
   INTEGER SUBROUTINE TIME(STRING)
   CHARACTER(LEN=8) STRING
   END FUNCTION TIME
END INTERFACE
```

| | |
|---|---|
| STRING | must be of type character and must provide at least 8 bytes of storage to contain the current time in the form: hh:mm:ss where *hh* is the current hour, *mm* the current minute, *ss* the number of seconds past the minute. |

### Description

This routine returns the system time in seconds since 00:00:00 GMT, January 1, 1970. It fills the STRING parameter with current time.

### Class

Nonstandard subroutine.

### Output

The elapsed time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

### Example

The following code sets the character variable tstr to the current system time (for example, 16:20:07).

```
CHARACTER(8) tstr
CALL TIME(tstr)
```

# TIMEF

*Returns the elapsed time since last called*

### Prototype

```
INTERFACE
    REAL FUNCTION TIMEF(TIME)
    INTEGER TIME
    END FUNCTION TIMEF
END INTERFACE
```

TIME              elapsed time in seconds

### Description

This function returns the number of seconds that elapsed since the first time TIMEF was called, or zero if the called for the first time.

### Output

The number of seconds that elapsed since the first time TIMEF was called, or zero if called for the first time.

# TOPEN

*Tape open*

### Prototype

```
INTERFACE
  INTEGER FUNCTION TOPEN(TLU, DEVNAME, LABELLED)
  INTEGER, INTENT(OUT) :: TLU
  CHARACTER(LEN=*), INTENT(IN) ::DEVNAME
  LOGICAL, INTENT (IN):: LABELLED
  END FUNCTION TOPEN
END INTERFACE
```

TLU             A Fortran logical unit number of 0 to 99 range

DEVNAME         the device name of the tape unit

LABELLED        indicates whether the tape to be opened is a labelled tape: 1 for a labeled tape or zero for an unlabeled tape

### Description

TOPEN opens a Fortran logical unit on a tape device for use with TREAD and TWRITE.

> **NOTE.** *This function is for Win32 systems only.*

### Output

0 for successful open or most recent value of errno for an error.

# TCLOSE

*Close a tape file*

### Prototype

```
INTERFACE
  INTEGER FUNCTION TCLOSE( TLU)
  INTEGER, INTENT(IN):: TLU
  END FUNCTION TCLOSE
END INTERFACE
```

TLU              a Win32 file handle descriptor

### Description

Using a file handle obtained from a previous call to TOPEN, TCLOSE closes a tape file on a tape drive on your local system.

> **NOTE.** *This function is for Win32 systems only.*

### Output

TCLOSE returns zero if the close operation was successful, or returns errno if the close operation was not successful.

# TREAD

*Read from a tape file*

### Prototype

```
INTERFACE
  INTEGER FUNCTION TREAD(TLU, BUFFER)
  INTEGER, INTENT(IN) :: TLU
  CHARACTER(LEN=*), INTENT(OUT) ::BUFFER
  LOGICAL, INTENT (IN):: LABELLED
  END FUNCTION TREAD
END INTERFACE
```

| TLU | is a file handle descriptor |
|---|---|
| BUFFER | a character variable, array, or array section large enough to hold the next record on the tape |

### Description

TREAD reads the next logical record from an already opened tape file. You must have previously opened the file using the TOPEN routine, and obtained a valid Win32 system file handle.

**NOTE.** *This function is for Win32 systems only.*

### Output

TREAD places the next logical record of a tape file in the input variable BUFFER. If the read operation is successful, TREAD returns zero as its result. If the read operation is not successful, TREAD returns errno as its result.

# TTYNAM

*Checks if the unit is a terminal*

### Prototype

```
INTERFACE
  CHARACTER(LEN=*) FUNCTION TTYNAM(LUN)
  INTEGER INTENT(IN)::LUN
  END FUNCTION TTYNAM
END INTERFACE
```

LUN                 a Fortran logical unit number

### Description

This function determines whether a particular logical unit is connected to a terminal (TTY) display device.

### Output

A string indicating the CHARACTER device name for a terminal device, or all blanks if not a terminal, or an error.

# TWRITE

*Writes to a tape file*

### Prototype

```
INTERFACE
  INTEGER FUNCTION TWRITE(TLU, BUFFER)
  INTEGER, INTENT(IN) :: TLU
  CHARACTER(LEN=*), INTENT(OUT) ::BUFFER
  END FUNCTION TREAD
END INTERFACE
```

| | |
|---|---|
| TLU | is a file handle descriptor |
| BUFFER | a CHARACTER expression whose value is data to be written to a tape file on your local system |

### Description

TWRITE takes the data you pass to it in the input expression BUFFER and writes it as a logical record to a tape device on your local system. You must have previously opened the tape device with a call to TOPEN, and obtained a valid Win32 system file handle.

### Output

TWRITE returns zero if the write operation was successful, and otherwise returns the system error code from errno.

## UNLINK

*Deletes a file by name*

### Prototype

```
INTERFACE
  INTEGER FUNCTION UNLINK (NAME)
  CHARACTER(LEN=*), INTENT(IN)::NAME
  END FUNCTION UNLINK
END INTERFACE
```

NAME     the name of the file you want to delete

### Description

UNLINK deletes a file with the name specified by NAME. You must have adequate permission to delete the file. The name can be any character expression that results in a valid file name. The name can include a full path name, including drive letter.

### Output

UNLINK returns a status code, which is zero if the deletion was successful, or the value of errno if the file deletion was not successful.

## UNPACKTIMEQQ

*Unpacks a packed time and date value.*

### Prototype

```
USE IFLPORT
or
INTERFACE
```

```
   SUBROUTINE UNPACKTIMEQQ(TIMEDATE,IYR, IMON,IDAY,
                             IHR,IMIN,ISEC)
      INTEGER(4) TIMEDATE
      INTEGER(2) IYR, IMON, IDAY, IHR, IMIN, ISEC
   END SUBROUTINE
 END INTERFACE
```

## Description

Unpacks a packed time and date value into its component parts. See
PACKTIMEQQ.

## Usage

```
CALL  UNPACKTIMEQQ(TIMEDATE,IYR,IMON,IDAY,
IHR,IMIN,ISEC)
```

TIMEDATE       `INTEGER(4)`. Packed time and date information.

IYR            `INTEGER(2)`. Year (*xxxx* AD).

IMON           `INTEGER(2)`. Month (1 - 12).

IDAY           `INTEGER(2)`. Day (1 - 31).

IHR            `INTEGER(2)`. Hour (0 - 23).

IMIN           `INTEGER(2)`. Minute (0 - 59).

ISEC           `INTEGER(2)`. Second (0 - 59).

`GETFILEINFOQQ` returns time and date in a packed format. You can use
`UNPACKTIMEQQ` to unpack these values. Use `PACKTIMEQQ` to repack times
for passing to `SETFILETIMEQQ`. Packed times can be compared using
relational operators.

## Example

```
   ! Note, compile this example with /nbs.
 USE IFLPORT
 CHARACTER(80)    file
 TYPE (FILE$INFO) info
 INTEGER(4) handle, result
 INTEGER(2) iyr, imon, iday, ihr, imin, isec
 file = 'd:\f90ps\bin\t???.*'
```

```
handle = FILE$FIRST
result = GETFILEINFOQQ(file, info, handle)
CALL UNPACKTIMEQQ(info%lastwrite, iyr, imon,&
                   iday, ihr, imin, isec)
WRITE(*,*) iyr, imon, iday
WRITE(*,*) ihr, imin, isec
END
```

# National Language Support Routines

National Language Support (NLS) procedures provide language localization and a subset of multi-byte character set (MBCS) NLS functions to let you write applications in different languages. To use an NLS routine, add the following statement to the program unit containing the procedure:

`USE IFLPORT`

or

`INCLUDE`
`"<installation directory>\perform\include\iflport.f90`

Table 2-1 summarizes the NLS procedures. The names are listed in mixed case to make the mnemonics easier to understand. When writing your applications, you can use any case.

**Table 2-1     Multi-byte Routines and Functions Summary**

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **Locale Setting and Inquiry** | | |
| **NLSEnumCodepages** <br> `ptr => NLSEnumCodepages ( )` | Function | Returns all the supported codepages on the system. |
| **NLSEnumLocales** <br> `ptr => NLSEnumLocales ( )` | Function | Returns all the languages and country combinations supported by the system. |
| **NLSGetEnvironmentCodepage** <br> `result =` <br> `NLSGetEnvironmentCodepage` <br> `( flags )` | Function | Returns the codepage number for the system (Window) codepage or the console codepage. |

**Table 2-1    Multi-byte Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **NLSGetLocale**<br>CALL NLSGetLocale (<br>[*language*] [, *country*]<br>[, *codepage* ] ) | Subroutine | Returns the current language, country, and codepage. |
| **NLSGetLocaleInfo**<br>result = NLSGetLocaleInfo<br>( *type, outstr* ) | Function | Returns requested information about the current local code se.t. |
| **NLSSetEnvironmentCodepage**<br>result =<br>NLSSetEnvironmentCodepage<br>( *codepage, flags* ) | Function | Changes the codepage for the current console. |
| **NLSSetLocale**<br>result = NLSSetLocale<br>( *language* [, *country*]<br>[, *codepage* ] ) | Function | Sets the language, country, and codepage. |

continued

| **Locale Formatting** | | |
|---|---|---|
| **NLSFormatCurrency**<br>result = NLSFormatCurrency<br>( *outstr, instr* [, *flags* ] ) | Function | Formats a number string and returns the correct currency string for the current locale. |
| **NLSFormatDate**<br>result = NLSFormatDate<br>( *outstr* [, *intime* ] [,<br>*flags* ] ) | Function | Returns a correctly formatted string containing the date for the current locale. |
| **NLSFormatNumber**<br>result = NLSFormatNumber<br>( *outstr, instr* [, *flags* ] ) | Function | Formats a number string and returns the correct number string for the current locale. |

**Table 2-1    Multi-byte Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **NLSFormatTime**<br>`result = NLSFormatTime`<br>`( outstr [, intime ]`<br>`[, flags ] )` | Function | Returns a correctly formatted string containing the time for the current locale. |
| **MBCS Inquiry** | | |
| **MBCharLen**<br>`result = MBCharLen (string)` | Function | Returns the length, in bytes, of the first character in a multi-byte-character string. |
| **MBCurMax**<br>`result = MBCurMax ( )` | Function | Returns the longest possible multi-byte character length, in bytes, for the current codepage. |
| **MBLead**<br>`result = MBLead ( char )` | Function | Determines whether a given character is the lead (first) byte of a multi-byte character sequence. |
| **MBLen**<br>`result = MBLen ( string )` | Function | Returns the number of characters in a multi-byte-character string, including trailing blanks. |
| | | continued |
| **MBLen_Trim**<br>`result = MBLen_Trim (string)` | Function | Returns the number of characters in a multi-byte-character string, not including trailing blanks. |
| **MBNext**<br>`result = MBNext ( string,`<br>`position )` | Function | Returns the position of the first lead byte or single-byte character immediately following the given position in a multi-byte-character string. |

**Table 2-1     Multi-byte Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **MBPrev**<br>result = MBPrev ( *string, position* ) | Function | Returns the position of the first lead byte or single-byte character immediately preceding the given string position in a multi-byte-character string. |
| **MBStrLead**<br>result = MBStrLead (*string, position*) | Function | Performs a context-sensitive test to determine whether a given character byte in a string is a multi-byte-character lead byte. |
| **MBCS Conversion** | | |
| **MBConvertMBToUnicode**<br>result = MBConvertMBToUnicode (*mbstr, unicodestr* [, *flags* ] ) | Function | Converts a character string from a multi-byte codepage to a Unicode string. |
| **MBConvertUnicodeToMB**<br>result = MBConvertUnicodeToMB (*unicodestr, mbstr* [,*flags*]) | Function | Converts a Unicode string to a multi-byte character string of the current codepage. |

continued

2-165

**Table 2-1    Multi-byte Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **MBCS Fortran Equivalent Procedures** | | |
| **MBINCHARQQ**<br>result = MBINCHARQQ (*string*) | Function | Same as INCHARQQ except that it can read a single multi-byte character at once and returns the number of bytes read. |
| **MBINDEX**<br>result = MBINDEX ( *string*, *substring* [, *back* ] ) | Function | Same as INDEX, except that multi-byte characters can be included in its arguments. |
| **MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE**<br>result = MBLGE ( *string_a*, *string_b*, [ *flags* ] ) | Function | Same as LGE, LGT, LLE, and LLT, and the logical operators .EQ. and .NE., except that multi-byte characters can be included in their arguments.<br>All these routines have the same arguments as shown for MBLGE. |
| **MBSCAN**<br>result = MBSCAN ( *string*, *set* [, *back* ] ) | Function | Same as SCAN, except that multi-byte characters can be included in its arguments. |
| **MBVERIFY**<br>result = MBVERIFY ( *string*, *set* [, *back* ] ) | Function | Same as VERIFY, except that multi-byte characters can be included in its arguments. |
| **MBJISTToJMS** | Function | Converts a Japan Industry Standard (JIS) character to a Microsoft Kanji (Shift JIS or JMS) character. |
| **MBJMSTToJIS** | Function | Converts a Microsoft Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character. |

## Locale Setting and Inquiry Procedures

# NLSEnumCodepages

*Returns all the supported codepages
on the system*

### Prototype

```
INTERFACE
  FUNCTION NLSenumCodepages()
    INTEGER(4), POINTER :: NLSEnumCodepages (:)
  END FUNCTION
END INTERFACE
```

### Description

Returns an array containing the code pages supported by the system, with each array element describing one valid codepage.

**NOTE.** *After use, the pointer returned by* NLSEnumCodepages *should be deallocated with the* DEALLOCATE *statement.*

### Output

Pointer to an array of codepages, with each element describing one supported codepage.

# NLSEnumLocales

*Returns all the language and country
combinations supported by the system*

### Prototype

```
INTERFACE
  FUNCTION NLSEnumLocales()
    INTEGER(4), PARAMETER :: NLS$MaxLanguageLen = 64
    INTEGER(4), PARAMETER :: NLS$MaxCountryLen = 64
    TYPE NLS$EnumLocale
      SEQUENCE
      CHARACTER(LEN= NLS$MaxLanguageLen) Language
      CHARACTER(LEN= NLS$MaxCountryLen) Country
      INTEGER(4) DefaultWindowsCodepage
      INTEGER(4) DefaultConsoleCodepage
    END TYPE
    TYPE(NLS$EnumLocale), POINTER::NLSEnumLocales (:)
  END FUNCTION
END INTERFACE
```

### Description

Returns an array containing the language and country combinations supported
by the system, in which each array element describes one valid combination.

**NOTE.** *After use, the pointer returned by* `NLSEnumLocales` *should
be deallocated with the* `DEALLOCATE` *statement.*

## Output

Pointer to an array of locales, in which each array element describes one supported language and country combination.

If the application is a Windows or a QuickWin application, `NLS$DefaultWindowsCodepage` is the codepage used by default for the given language and country combination. If the application is a console application, `NLS$DefaultConsoleCodepage` is the codepage used by default for the given language and country combination.

# NLSGetEnvironmentCodepage

*Returns the codepage number for the system or the console codepage*

### Prototype

```
INTERFACE

  INTEGER(4) FUNCTION
NLSGetEnvironmentCodepage(FLAGS)

  INTEGER(4), INTENT(IN) :: FLAGS

  END FUNCTION

END INTERFACE
```

FLAGS             Tells the function which codepage number to return. Available values are:

`NLS$ConsoleEnvironmentCodepage` - gets the codepage for the console

`NLS$WindowsEnvironmentCodepage` - gets the current Windows codepage

## Description

Returns the codepage number for the system (Window) codepage or the console codepage.

**Output**

Zero if successful; otherwise, returns one of the following error codes:

`NLS$ErrorInvalidFlags`

indicates that FLAGS has an illegal value

`NLS$ErrorNoConsole`

there is no console associated with the given application; therefore, operations with the console codepage are not possible.

# NLSGetLocale

*Returns the current language, country, and/or codepage.*

**Prototype**

```
INTERFACE
  SUBROUTINE NLSGetLocale (LANGUAGE, COUNTRY, &
                          CODEPAGE)
  CHARACTER(LEN=*), INTENT(OUT), OPTIONAL::LANGUAGE
  CHARACTER(LEN=*), INTENT(OUT), OPTIONAL::COUNTRY
  INTEGER(4), INTENT(OUT), OPTIONAL::CODEPAGE
  END SUBROUTINE
END INTERFACE
```

`LANGUAGE`    Optional, output. Character(LEN=*). Current language.

`COUNTRY`    Optional, output. Character(LEN=*). Current country.

`CODEPAGE`    Optional, output. Character(LEN=*). Current codepage.

**Description**

Retrieves the current language, country, and/or codepage.

> **NOTE.** `NLSGetLocale` *returns a valid codepage in codepage. It does not return one of the* `NLS$...` *symbolic constants that can be used with* `NLSSetLocale`*.*

### Output

Requested information about the current language, country, and codepage.

# NLSGetLocaleInfo

*Returns information about the current local code set*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION NLSGetLocaleInfo(INFOTYPE, &
                                        OUTSTR)
    INTEGER(4), INTENT(IN)::INFOTYPE
    CHARACTER(LEN=*), INTENT(OUT)::OUTSTR
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| INFOTYPE | Input  NLS parameter requested. A list of parameter names is given in the NLS Locale Info Parameters in "Note." |
| OUTSTR | Output. Character(LEN=*). Parameter setting for the current locale. All parameter settings placed in `OUTSTR` are character strings, even numbers. If a parameter setting is numeric, the ASCII representation of the number is used. If |

the requested parameter is a date or time string, an explanation of how to interpret the format in `OUTSTR` is given in NLS Date and Time Format.

## Description

Returns requested information about the current local code set.

---

**NOTE.** *The* `NLS$LI` *parameters are used for the argument* `INFOTYPE` *and select the locale information returned by* `NLSGetLocaleInfo` *in* `OUTSTR`. *You can perform an inclusive OR with* `NLS$NoUserOverride` *and any* `NLS$LI` *parameter. This causes* `NLSGetLocaleInfo` *to bypass any user overrides and always return the system default value.*

---

Table 2-2 lists and describes all NLS$LI parameters.

**Table 2-2       NLS$LI Parameters**

| Name | Description |
| --- | --- |
| NLS$LI_ILANGUAGE | An ID indicating the language |
| NLS$LI_SLANGUAGE | The full localized name of the language. |
| NLS$LI_SENGLANGUAGE | The full English name of the language from the ISO Standard 639. This is limited to characters that map into the ASCII 127 character subset. |
| NLS$LI_SABBREVLANGNAME | The abbreviated name of the language, created by taking the 2-letter language abbreviation as found in ISO Standard 639 and adding a third letter as appropriate to indicate the sublanguage. |
| NLS$LI_SNATIVELANGNAME | The native name of the language. |

continued

**Table 2-2     NLS$LI Parameters** (continued)

| Name | Description |
|---|---|
| NLS$LI_ICOUNTRY | The country code, based on international phone codes, also referred to as IBM country codes. |
| NLS$LI_SCOUNTRY | The full localized name of the country. |
| NLS$LI_SENGCOUNTRY | The full English name of the country. This will always be limited to characters that map into the ASCII 127 character subset. |
| NLS$LI_SABBREVCTRYNAME | The abbreviated name of the country as per ISO Standard 3166. |
| NLS$LI_SNATIVECTRYNAME | The native name of the country. |
| NLS$LI_IDEFAULTLANGUAGE | Language ID for the principal language spoken in this locale. This is provided so that partially specified locales can be completed with default values. |
| NLS$LI_IDEFAULTCOUNTRY | Country code for the principal country in this locale. This is provided so that partially specified locales can be completed with default values. |
| NLS$LI_IDEFAULTANSICODEPAGE | ANSI code page associated with this locale. |
| NLS$LI_IDEFAULTOEMCODEPAGE | OEM code page associated with the locale. |
| NLS$LI_SLIST | Character(s) used to separate list items, for example, comma in many locales. |
| NLS$LI_IMEASURE | This value is 0 if the metric system (S.I.) is used and 1 for the U.S. system of measurements. |

continued

**Table 2-2     NLS$LI Parameters** (continued)

| Name | Description |
|------|-------------|
| NLS$LI_SDECIMAL | The character(s) used as decimal separator. This cannot be set to digits 0 - 9. |
| NLS$LI_STHOUSAND | The character(s) used as separator between groups of digits left of the decimal. This cannot be set to digits 0 - 9. |
| NLS$LI_SGROUPING | Sizes for each group of digits to the left of the decimal. An explicit size is needed for each group; sizes are separated by semicolons. If the last value is 0 the preceding value is repeated. To group thousands, specify "3;0". |
| NLS$LI_IDIGITS | The number of decimal digits. |
| NLS$LI_IDIGITSNLS$LI_ILZERO | Determines whether to use leading zeros in decimal fields:<br>0 - Use no leading zeros<br>1 - Use leading zeros. |
| NLS$LI_INEGNUMBER | Determines how negative numbers are represented:<br>0 - Puts negative numbers in parentheses: (1.1)<br>1 - Puts a minus sign in front: -1.1<br>2 - Puts a minus sign followed by a space in front: - 1.1<br>3 - Puts a minus sign after: 1.1-<br>4 - Puts a space then a minus sign after: 1.1 - |

**Table 2-2 NLS$LI Parameters** (continued)

| Name | Description |
|------|-------------|
| NLS$LI_SNATIVEDIGITS | The ten characters that are the native equivalent to the ASCII 0-9. |
| NLS$LI_SCURRENCY | The string used as the local monetary symbol. Cannot be set to digits 0-9. |
| NLS$LI_SINTLSYMBOL | Three characters of the International monetary symbol specified in ISO 4217 "Codes for the Representation of Currencies and Funds", followed by the character separating this string from the amount. |
| NLS$LI_SMONDECIMALSEP | The character(s) used as monetary decimal separator. This cannot be set to digits 0-9. |
| NLS$LI_SMONTHOUSANDSEP | The character(s) used as monetary separator between groups of digits left of the decimal. Cannot be set to digits 0-9. |
| NLS$LI_SMONGROUPING | Sizes for each group of monetary digits to the left of the decimal. If the last value is 0, the preceding value is repeated. To group thousands, specify "3;0". |
| NLS$LI_ICURRDIGITS | Number of decimal digits for the local monetary format. |
| NLS$LI_IINTLCURRDIGITS | Number of decimal digits for the international monetary format. |

continued

**Table 2-2     NLS$LI Parameters** (continued)

| Name | Description |
|------|-------------|
| NLS$LI_ICURRENCY | Determines how positive currency is represented:<br>0 - Puts currency symbol in front with no separation: $1.1<br>1 - Puts currency symbol in back with no separation: 1.1$<br>2 - Puts currency symbol in front with single space after: $ 1.1<br>3 - Puts currency symbol in back with single space before: 1.1 $ |
| NLS$LI_INEGCURR | Determines how negative currency is represented:<br>0:  ($1.1)<br>1:  -$1.1<br>2:  $-1.1<br>3:  $1.1-<br>4:  (1.1$)<br>5:  -1.1$<br>6:   1.1-$<br>7:   1.1$-<br>8:  -1.1 $ (space before $)<br>9:  -$ 1.1 (space after $)<br>10:  1.1 $- (space before $)<br>11:  $ 1.1- (space after $)<br>12:  $ -1.1 (space after $)<br>13:  1.1- $ (space before $)<br>14:  ($ 1.1) (space after $)<br>15:  (1.1 $) (space before $) |

**Table 2-2    NLS$LI Parameters** (continued)

| Name | Description |
|------|-------------|
| NLS$LI_SPOSITIVESIGN | String value for the positive sign. Cannot be set to digits 0-9. |
| NLS$LI_SNEGATIVESIGN | String value for the negative sign. Cannot be set to digits 0-9. |
| NLS$LI_IPOSSIGNPOSN | Determines the formatting index for positive values:<br>0 - Parenthesis surround the amount and the monetary symbol<br>1 - The sign string precedes the amount and the monetary symbol<br>2 - The sign string follows the amount and the monetary symbol<br>3 - The sign string immediately precedes the monetary symbol<br>4 - The sign string immediately follows the monetary symbol |
| NLS$LI_INEGSIGNPOSN | Determines the formatting index for negative values. Same values as for NLS$LI_IPOSSIGNPOSN |
| NLS$LI_IPOSSYMPRECEDES | 1 if the monetary symbol precedes, 0 if it follows a positive amount. |
| NLS$LI_IPOSSEPBYSPACE | 1 if the monetary symbol is separated by a space from a positive amount, 0 otherwise. |
| NLS$LI_INEGSYMPRECEDES | 1 if the monetary symbol precedes, 0 if it follows a negative amount |
| NLS$LI_INEGSEPBYSPACE | 1 if the monetary symbol is separated by a space from a negative amount, 0 otherwise. |

continued

**Table 2-2    NLS$LI Parameters** (continued)

| Name | Description |
|------|-------------|
| NLS$LI_STIMEFORMAT | Time formatting string. See the NLS Date and Time Format section for explanations of the valid strings. |
| NLS$LI_STIME | Character(s) for the time separator. Cannot be set to digits 0-9. |
| NLS$LI_ITIME | Time format:<br>0 - Use 12-hour format<br>1 - Use 24-hour format |
| NLS$LI_ITLZERO | Determines whether to use leading zeros in time fields:<br>0 - Use no leading zeros<br>1 - Use leading zeros for hours |
| NLS$LI_S1159 | String for the AM designator |
| NLS$LI_S2359 | String for the PM designator. |
| NLS$LI_SSHORTDATE | Short Date formatting string for this locale. The d, M and y should have the day, month, and year substituted, respectively. See the NLS Date and Time Format section for explanations of the valid strings. |
| NLS$LI_SDATE | Character(s) for the date separator. Cannot be set to digits 0-9. |
| NLS$LI_IDATE | Short Date format ordering:<br>0 - Month-Day-Year<br>1 - Day-Month-Year<br>2 - Year-Month-Day |

**Table 2-2    NLS$LI Parameters** (continued)

| Name | Description |
|------|-------------|
| NLS$LI_ICENTURY | Specifies whether to use full 4-digit century for the short date only:<br>0 - Two-digit year<br>1 - Full century |
| NLS$LI_IDAYLZERO | Specifies whether to use leading zeros in day fields for the short date only:<br>0 - Use no leading zeros<br>1 - Use leading zeros |
| NLS$LI_IMONLZERO | Specifies whether to use leading zeros in month fields for the short date only:<br>0 - Use no leading zeros<br>1 - Use leading zeros |
| NLS$LI_SLONGDATE | Long Date formatting string for this locale. The string returned may contain a string within single quotes (' '). Any characters within single quotes should be left as is. The d, M and y should have the day, month, and year substituted, respectively. |
| NLS$LI_ILDATE | Long Date format ordering:<br>0 - Month-Day-Year<br>1 - Day-Month-Year<br>2 - Year-Month-Day |
| NLS$LI_ICALENDARTYPE | Specifies which type of calendar is currently being used:<br>1 - Gregorian (as in United States)<br>2 - Gregorian (English strings always)<br>3 - Era: Year of the Emperor (Japan)<br>4 - Era: Year of the Republic of China<br>5 - Tangun Era (Korea) |

**Table 2-2      NLS$LI Parameters** (continued)

| Name | Description |
|---|---|
| NLS$LI_IOPTIONALCALENDAR | Specifies which additional calendar types are valid and available for this locale. This can be a null separated list of all valid optional calendars:<br>0 - No additional types valid<br>1 - Gregorian (localized)<br>2 - Gregorian (English strings always)<br>3 - Era: Year of the Emperor (Japan)<br>4 - Era: Year of the Republic of China<br>5 - Tangun Era (Korea) |
| NLS$LI_IFIRSTDAYOFWEEK | Specifies which day is considered first in a week:<br>0 - SDAYNAME1<br>1 - SDAYNAME2<br>2 - SDAYNAME3<br>3 - SDAYNAME4<br>4 - SDAYNAME5<br>5 - SDAYNAME6<br>6 - SDAYNAME7 |
| NLS$LI_IFIRSTWEEKOFYEAR | Specifies which week of the year is considered first:<br>0 - Week containing 1/1<br>1 - First full week following 1/1<br>2 - First week containing at least 4 days |
| NLS$LI_SDAYNAME1 | Long name for Monday |
| NLS$LI_SDAYNAME2 | Long name for Tuesday |
| NLS$LI_SDAYNAME3 | Long name for Wednesday |
| NLS$LI_SDAYNAME4 | Long name for Thursday |
| NLS$LI_SDAYNAME5 | Long name for Friday |
| NLS$LI_SDAYNAME6 | Long name for Saturday |
| NLS$LI_SDAYNAME7 | Long name for Sunday |
| NLS$LI_SABBREVDAYNAME1 | Abbreviated name for Monday |
| NLS$LI_SABBREVDAYNAME2 | Abbreviated name for Tuesday |
| NLS$LI_SABBREVDAYNAME3 | Abbreviated name for Wednesday |

**Table 2-2      NLS$LI Parameters** (continued)

| Name | Description |
|------|-------------|
| NLS$LI_SABBREVDAYNAME4 | Abbreviated name for Thursday |
| NLS$LI_SABBREVDAYNAME5 | Abbreviated name for Friday |
| NLS$LI_SABBREVDAYNAME6 | Abbreviated name for Saturday |
| NLS$LI_SABBREVDAYNAME7 | Abbreviated name for Sunday |
| NLS$LI_SMONTHNAME1 | Long name for January |
| NLS$LI_SMONTHNAME2 | Long name for February |
| NLS$LI_SMONTHNAME3 | Long name for March |
| NLS$LI_SMONTHNAME4 | Long name for April |
| NLS$LI_SMONTHNAME5 | Long name for May |
| NLS$LI_SMONTHNAME6 | Long name for June |
| NLS$LI_SMONTHNAME7 | Long name for July |
| NLS$LI_SMONTHNAME8 | Long name for August |
| NLS$LI_SMONTHNAME9 | Long name for September |
| NLS$LI_SMONTHNAME10 | Long name for October |
| NLS$LI_SMONTHNAME11 | Long name for November |
| NLS$LI_SMONTHNAME12 | Long name for December |
| NLS$LI_SMONTHNAME13 | Long name for 13th month (if exists) |
| NLS$LI_SABBREVMONTHNAME1 | Abbreviated name for January |
| NLS$LI_SABBREVMONTHNAME2 | Abbreviated name for February |
| NLS$LI_SABBREVMONTHNAME3 | Abbreviated name for March |
| NLS$LI_SABBREVMONTHNAME4 | Abbeviated name for April |
| NLS$LI_SABBREVMONTHNAME5 | Abbreviated name for May |
| NLS$LI_SABBREVMONTHNAME6 | Abbreviated name for June |
| NLS$LI_SABBREVMONTHNAME7 | Abbreviated name for July |
| NLS$LI_SABBREVMONTHNAME8 | Abbreviated name for August |
| NLS$LI_SABBREVMONTHNAME9 | Abbreviated name for September |
| NLS$LI_SABBREVMONTHNAME10 | Abbreviated name for October |
| NLS$LI_SABBREVMONTHNAME11 | Abbreviated name for November |

**Table 2-2      NLS$LI Parameters** (continued)

| Name | Description |
| --- | --- |
| NLS$LI_SABBREVMONTHNAME12 | Abbreviated name for December |
| NLS$LI_SABBREVMONTHNAME13 | Abbreviated name for 13th month (if exists) |

### Output

Returns the number of characters written to OUTSTR if successful. If OUTSTR has 0 length, the number of characters required to hold the requested information is returned. Otherwise, one of the following error codes returns:

NLS$ErrorInvalidLIType

The given INFOTYPE is invalid.

NLS$ErrorInsufficientBuffer

The OUTSTR buffer was too small, but was not 0 (so that the needed size would be returned).

## NLSSetEnvironmentCodepage

*Changes the codepage for the current console*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION
       NLSSetEnvironmentCodepage(CODEPAGE, FLAGS)
  INTEGER(4), INTENT(IN) :: CODEPAGE
  INTEGER(4), INTENT(IN) :: FLAGS
  END FUNCTION
END INTERFACE
```

### Description

Sets the codepage for the current console. The specified codepage affects the current console program and any other programs launched from the same console. It does not affect other open consoles or any consoles opened later.

> **NOTE.** *The* `FLAGS` *argument must be* `NLS$ConsoleEnvironmentCodepage`*; it cannot be* `NLS$WindowsEnvironmentCodepage`*.* `NLSSetEnvironmentCodepage` *does not affect the Windows codepage*

### Output

Returns zero if successful. Otherwise, returns one of the following error codes:

`NLS$ErrorInvalidCodepage`
  `CODEPAGE` is invalid or not installed on the system

`NLS$ErrorInvalidFlags`
  `FLAGS` is not valid.

`NLS$ErrorNoConsole`
  There is no console associated with the given application; therefore operations, with the console codepage are not possible.

# NLSSetLocale

*Sets the language, country, and codepage*

### Prototype

```
INTERFACE

  INTEGER(4) FUNCTION NLSSetLocale(LANGUAGE,COUNTRY,&
CODEPAGE)

    CHARACTER(LEN=*), INTENT(IN) :: LANGUAGE

    CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: COUNTRY

    INTEGER(4), INTENT(IN), OPTIONAL :: CODEPAGE

  END FUNCTION

END INTERFACE
```

LANGUAGE        Input. `CHARACTER(LEN=*)`. One of the languages supported by the Win32* NLS APIs.

COUNTRY         Optional, input. `CHARACTER(LEN=*)`. If specified, characterizes the language further. If omitted, the default country for the language is set.

CODEPAGE        Optional, input. `INTEGER(4)`. If specified, codepage to use for all character-oriented NLS functions. Can be any valid supported codepage or one of the following predefined values:

`NLS$CurrentCodepage`
                The codepage is not changed. Only the language and country settings are altered by the function.

`NLS$ConsoleEnvironmentCodepage`
                The codepage is changed to the default environment codepage currently in effect for console programs.

NLS$ConsoleLanguageCodepage

> The codepage is changed to the default console codepage for the language and country combination specified.

NLS$WindowsEnvironmentCodepage

> The codepage is changed to the default environment codepage currently in effect for Windows programs.

NLS$WindowsLanguageCodepage

> The codepage is changed to the default Windows codepage for the language and country combination specified.

If you omit CODEPAGE, it defaults to NLS$WindowsLanguageCodepage. At program startup, NLS$WindowsEnvironmentCodepage is used to set the codepage.

### Description

Sets the current language, country, and/or codepage.

**NOTE.** NLSSetLocale *works on installed locales only. Windows NT and Windows 95 support many locales, but these must be installed through the system Windows NT Control Panel/International menu or the Windows 95 Control Panel/Regional Settings menu.*

In addition to the note above take into consideration the following:

- When doing mixed-language programming with Fortran and C, calling NLSSetLocale with a codepage other than the default environment Windows codepage causes the codepage in the C run-time library to change by calling C language setmbcp( ) routine with the new codepage. Conversely, changing the C run-time library codepage does not change the codepage in the Fortran NLS library

- Calling NLSSetLocale has no effect on the locale used by C programs. The locale set with C language setlocale( ) routine is independent of NLSSetLocale.

- Calling `NLSSetLocale` with the default environment console codepage, `NLS$ConsoleEnvironmentCodepage`, causes an implicit call to the Win32 API `SetFileApisToOEM( )`. Calling `NLSSetLocale` with any other codepage causes a call to `SetFileApisToANSI( )`.

**Output**

Zero if successful. Otherwise, one of the following error codes may be returned:

`NLS$ErrorInvalidLanguage`
> `LANGUAGE` is invalid or not supported.

`NLS$ErrorInvalidCountry`
> `COUNTRY` is invalid or is not valid with the language specified.

`NLS$ErrorInvalidCodepage`
> `CODEPAGE` is invalid or not installed on the system.

## Locale Formatting Procedures

## NLSFormatCurrency

*Returns a correctly formatted*
*currency string for the current locale*

**Prototype**

```
INTERFACE
  INTEGER(4) FUNCTION NLSFormatCurrency(OUTSTR, &
                                  INSTR, FLAGS)
  INTEGER(4), INTENT(IN), OPTIONAL :: FLAGS
  CHARACTER(LEN=*), INTENT(IN) :: INSTR
    CHARACTER(LEN=*), INTENT(OUT) :: OUTSTR
```

```
    END FUNCTION
END INTERFACE
```

OUTSTR      Output. `CHARACTER(LEN=*)`. String containing the
            correctly formatted currency for the current locale. If
            `OUTSTR` is longer than the formatted currency, it is
            blank-padded.

INSTR       Input. `CHARACTER(LEN=*)`. Number string to be
            formatted. Can contain only the characters 0' through
            9', one decimal point (a period) if a floating-point
            value, and a minus sign in the first position if
            negative. All other characters are invalid and cause
            the function to return an error.

FLAGS       Optional, input. `INTEGER(4)`. If specified, modifies
            the currency conversion. If you omit `FLAGS` *s*, the
            flag `NLS$Normal` is used. Available values are:

            `NLS$Normal`
                              No special formatting

            `NLS$NoUserOverride`
                              Do not use user overrides

### Description

Formats a number string and returns the correct currency string for the
current locale.

### Output

Number of characters written to `OUTSTR` (bytes are counted, not
multibyte characters), or one of the following negative values if an error
occurs:

`NLS$ErrorInsufficentBuffer`
                  `OUTSTR` buffer is too small

`NLS$ErrorInvalidInput`
                  `FLAGS` has an illegal value

`NLS$ErrorInvalidFlags`
                  `INSTR` has an illegal value.

# NLSFormatDate

*Returns a correctly formatted string containing the date for the current locale*

## Prototype

```
INTERFACE
  INTEGER(4) FUNCTION NLSFormatDate(OUTSTR, INTIME, &
                                    FLAGS)
  INTEGER(4), INTENT(IN), OPTIONAL :: FLAGS, INTIME
  CHARACTER(LEN=*), INTENT(OUT) :: OUTSTR
  END FUNCTION
END INTERFACE
```

OUTSTR          Output. `CHARACTER(LEN=*)`. String containing the correctly formatted date for the current locale. If `OUTSTR` is longer than the formatted date, it is blank-padded.

INTIME          Optional, input. `INTEGER(4)`. If specified, date to be formatted for the current locale. Must be an integer date such as the packed time created with `PACKTIMEQQ`. If you omit `INTIME`, the current system date is formatted and returned in `OUTSTR`.

FLAGS           Optional, input. `INTEGER(4)`. If specified, modifies the date conversion. If you omit `FLAGS`, the flag `NLS$Normal` is used. Available values are:

`NLS$Normal`   No special formatting

`NLS$NoUserOverride`
                Do not use user overrides

`NLS$UseAltCalendar`
                Use the locale's alternate calendar

<div align="center">

NLS$LongDate

Use local long date format

NLS$ShortDate

Use local short date format

</div>

### Description

Returns a correctly formatted string containing the date for the current locale.

### Output

Number of characters written to OUTSTR (bytes are counted, not multibyte characters), or one of the following negative values if an error occurs:

NLS$ErrorInsufficentBuffer

OUTSTR buffer is too small

NLS$ErrorInvalidInput

INTIME has an illegal value

NLS$ErrorInvalidFlags

FLAGS has an illegal value

# NLSFormatNumber

*Returns a correctly formatted number string for the current locale*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION NLSFormatNumber(OUTSTR, INSTR,&
                                      FLAGS)
   INTEGER(4), INTENT(IN), OPTIONAL :: FLAGS
    CHARACTER(LEN=*), INTENT(IN) :: INSTR
```

```
    CHARACTER(LEN=*), INTENT(OUT) :: OUTSTR
  END FUNCTION
END INTERFACE
```

OUTSTR
Output. `CHARACTER(LEN=*)`. String containing the correctly formatted number for the current locale. If `OUTSTR` is longer than the formatted number, it is blank-padded.

INSTR
Input. `CHARACTER(LEN=*)`. Number string to be formatted. Can only contain the characters 0' through 9', one decimal point (a period) if a floating-point value, and a minus sign in the first position if negative. All other characters are invalid and cause the function to return an error.

FLAGS
Optional, input. `INTEGER(4)`. If specified, modifies the number conversion. If you omit `FLAGS`, the flag `NLS$Normal` is used. Available values are:

`NLS$Normal`
No special formatting

`NLS$NoUserOverride`
Do not use user overrides

## Description

Formats a number string and returns the correct number string for the current locale.

## Output

Number of characters written to `OUTSTR` (bytes are counted, not multibyte characters), or one of the following negative values if an error occurs:

`NLS$ErrorInsufficentBuffer`
`OUTSTR` buffer is too small

`NLS$ErrorInvalidInput`
`INSTR` has an illegal value

```
NLS$ErrorInvalidFlags
            FLAGS has an illegal value
```

# NLSFormatTime

*Returns a correctly formatted string
containing the time for the current
locale*

## Prototype

```
INTERFACE
  INTEGER(4) FUNCTION NLSFormatTime(OUTSTR, INTIME, &
                                    FLAGS)
  INTEGER(4), INTENT(IN), OPTIONAL :: FLAGS, INTIME
  CHARACTER(LEN=*), INTENT(OUT) :: OUTSTR
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| OUTSTR | Output. `CHARACTER(LEN=*)`. String containing the correctly formatted time for the current locale. If `OUTSTR` is longer than the formatted time, it is blank-padded. |
| INTIME | Optional, input. `INTEGER(4)`. If specified, time to be formatted for the current locale. Must be an integer time such as the packed time created with `PACKTIMEQQ`. If you omit `INTIME`, the current system time is formatted and returned in `OUTSTR`. |
| FLAGS | Optional, input. `INTEGER(4)`. If specified, modifies the time conversion. If you omit `FLAGS`, the flag `NLS$Normal` is used. Available values are: |

`NLS$Normal`        No special formatting

`NLS$NoUserOverride`

                         Do not use user overrides

NLS$NoMinutesOrSeconds
Do not return minutes or seconds

NLS$NoSeconds  Do not return seconds

NLS$NoTimeMarker
Do not add a time marker string

NLS$Force24HourFormat
Return string in 24 hour format

### Description

Returns a correctly formatted string containing the time for the current locale.

### Output

Number of characters written to OUTSTR (bytes are counted, not multibyte characters), or one of the following negative values if an error occurs:

NLS$ErrorInsufficentBuffer
OUTSTR buffer is too small

NLS$ErrorInvalidInput
INTIME has an illegal value

NLS$ErrorInvalidFlags
FLAGS has an illegal value

# MBCS Inquiry Procedures

## MBCharLen

*Returns the length, in bytes, of the first character in a multibyte-character string*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION MBCharLen(STRING)
  CHARACTER(LEN=*), INTENT(IN) :: STRING
  END FUNCTION
END INTERFACE
```

STRING            Input. `CHARACTER(LEN=*)`. String containing the character whose length is to be determined. Can contain multibyte characters.

### Description

Returns the length, in bytes, of the first character in a multibyte-character string.

**NOTE.** `MBCharLen` *does not test for multibyte character validity.*

### Output

Number of bytes in the first character contained in *string*. Returns 0 if `STRING` has no characters (is length 0).

# MBCurMax

*Returns the longest possible multibyte character length, in bytes, for the current codepage*

## Prototype

```
INTERFACE
   INTEGER(4) FUNCTION MBCurMax()
   END FUNCTION
END INTERFACE
```

## Description

Returns the longest possible multibyte character length, in bytes, for the current codepage.

**NOTE.** *The* `MBLenMax` *parameter, defined in the module IFLPORT.F90, is the longest length, in bytes, of any character in any codepage installed on the system.*

## Output

Longest possible multibyte character, in bytes, for the current codepage.

# MBLen

*Returns the number of characters in a multibyte-character string, including trailing blanks*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION MBLen(STRING)
  CHARACTER(LEN=*), INTENT(IN) :: STRING
  END FUNCTION
END INTERFACE
```

STRING          Input. CHARACTER(LEN=*). String whose characters are to be counted. Can contain multibyte characters.

### Description

Returns the number of characters in a multibyte-character string, including trailing blanks.

**NOTE.** MBLen *recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity*

### Output

Number of characters in STRING.

# MBLen_Trim

*Returns the number of characters in a multibyte-character string, not including trailing blanks*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION MBLen_Trim(STRING)
  CHARACTER(LEN=*), INTENT(IN) :: STRING
  END FUNCTION
END INTERFACE
```

STRING          Input. `CHARACTER(LEN=*)`. String whose characters are to be counted. Can contain multibyte characters.

### Description

Returns the number of characters in a multibyte-character string, not including trailing blanks.

**NOTE.**  `MBLen_Trim` *recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity.*

### Output

Number of characters in `STRING` minus any trailing blanks (blanks are bytes containing character 32 (hex 20) in the ASCII collating sequence).

# MBNext

*Returns the position of the first lead byte or single-byte character immediately following the given position in a multibyte-character string*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION MBNext(STRING, POSITION)
  CHARACTER(LEN=*), INTENT(IN) :: STRING
  INTEGER(4), INTENT(IN) :: POSITION
  END FUNCTION
END INTERFACE
```

STRING       Input. `CHARACTER(LEN=*)`. String to be seached for the first lead byte or single-byte character after the current position. Can contain multibyte characters.

POSITION       Input. `INTEGER(4)`. Position in `STRING` to search from. Must be the position of a lead byte or a single-byte character. Cannot be the position of a trail (second) byte of a multibyte character.

### Description

Returns the position of the first lead byte or single-byte character immediately following the given position in a multibyte-character string.

### Output

Position of the first lead byte or single-byte character in `STRING` immediately following the position given in `POSITION`, or 0 if no following first byte is found in `STRING`.

# MBPrev

*Returns the position of the first lead byte or single-byte character immediately preceding the given string position in a multibyte-character string*

## Prototype

```
INTERFACE
  INTEGER(4) FUNCTION MBPrev(STRING, POSITION)
  CHARACTER(LEN=*), INTENT(IN) :: STRING
  INTEGER(4), INTENT(IN) :: POSITION
  END FUNCTION
END INTERFACE
```

STRING          Input. `CHARACTER(LEN=*)`. String to be seached for the first lead byte or single-byte character before the current position. Can contain multibyte characters.

POSITION        Input. `INTEGER(4)`. Position in `STRING` to search from. Must be the position of a lead byte or single-byte character. Cannot be the position of the trail (second) byte of a multibyte character.

## Description

Returns the position of the first lead byte or single-byte character immediately preceding the given string position in a multibyte-character string.

## Output

Position of the first lead byte or single-byte character in `STRING` immediately preceding the position given in `POSITION`, or 0 if no preceding first byte is found in `STRING`.

# MBStrLead

*Performs a context-senstive test to determine whether a given character byte in a string is a multibyte-character lead byte*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION MBStrLead(STRING, POSITION)
  CHARACTER(LEN=*), INTENT(IN) :: STRING
  INTEGER(4), INTENT(IN) :: POSITION
  END FUNCTION
END INTERFACE
```

STRING          Input. `CHARACTER(LEN=*)`. String containing the character byte to be tested for lead status.

POSITION        Input. `INTEGER(4)`. Position in `STRING` of the character byte in the string to be tested.

### Description

Performs a context-senstive test to determine whether a given character byte in a string is a multibyte-character lead byte.

> **NOTE.** `MBStrLead` *is passed a whole string and can identify any byte within the string as a lead or trail byte because it performs a context-sensitive test, scanning all the way back to the beginning of a string if necessary to establish context.* `MBLead` *is passed only one character at a time and must start on a lead byte and step through a string one character at a time to establish context for the character. Thus,* `MBStrLead` *can be much slower than* `MBLead` *(up to* n *times slower, where* n *is the length of the string).*

**Output**

Returns `.TRUE.` if the character byte in `POSITION` of `STRING` is a lead byte; otherwise, `.FALSE.`.

# MBCS Conversion Procedures

# MBConvertMBToUnicode

*Converts a character string from a multi-byte codepage to a Unicode string*

**Prototype**

```
INTERFACE
  INTEGER(4) FUNCTION MBConvertMBToUnicode(MBSTR,&
                                  UNICODESTR, FLAGS)
  CHARACTER(LEN=*), INTENT(IN) :: MBSTR
  INTEGER(2), DIMENSION(:), INTENT(OUT) :: UNICODESTR
  INTEGER(4), INTENT(IN), OPTIONAL :: FLAGS
  END FUNCTION
END INTERFACE
```

MBSTR            Input. `CHARACTER(LEN=*)`. Multibyte codepage string to be converted.

UNICODESTR       Output. `INTEGER(2)`. Array of integers that is the translation of the input string into Unicode.

FLAGS            Optional, input. `INTEGER(4)`. If specified, modifies the string conversion. If `FLAGS` is omitted, the value `NLS$Precomposed` is used. Available values are:

                 `NLS$Precomposed`
                                  Use precomposed characters always. (default)

`NLS$Composite`
> Use composite wide characters always.

`NLS$UseGlyphChars`
> Use glyph characters instead of control characters.

`NLS$ErrorOnInvalidChars`
> Returns - 1 if an invalid input character is encountered.

The flags `NLS$Precomposed` and `NLS$Composite` are mutually exclusive. You can combine `NLS$UseGlyphChars` with either `NLS$Precomposed` or `NLS$Composite` using an inclusive OR (`IOR` or `OR`).

### Description

Converts a multibyte-character string from the current codepage to a Unicode string.

---

**NOTE.** *By default, or if `FLAGS` is set to **NLS$Precomposed**, the function **MBConvertMBToUnicode** attempts to translate the multibyte codepage string to a precomposed Unicode string. If a precomposed form does not exist, the function attempts to translate the codepage string to a composite form.*

---

### Output

If no error occurs, returns the number of bytes written to `UNICODESTR` (bytes are counted, not characters), or the number of bytes required to hold the output string if `UNICODESTR` has zero size. If the `UNICODESTR` array is bigger than needed to hold the translation, the extra elements are set to 0. If `UNICODESTR` has zero size, the function returns the number of bytes required to hold the translation and nothing is written to `UNICODESTR`.

If an error occurs, one of the following negative values is returned:

NLS$ErrorInsufficentBuffer

> The UNICODESTR argument is too small, but not zero size so that the needed number of bytes would be returned.

NLS$ErrorInvalidFlags

> The FLAGS argument has an illegal value.

NLS$ErrorInvalidCharacter

> A character with no Unicode translation was encountered in MBSTR. This error can occur only if the NLS$InvalidCharsError flag was used in FLAGS.

# MBConvertUnicodeToMB

*Converts a Unicode string to a multi-byte character string of the current codepage*

## Prototype

```
INTERFACE
  INTEGER(4) FUNCTION
   MBConvertUnicodeToMB(UNICODESTR, MBSTR, FLAGS)
  INTEGER(2), DIMENSION(:), INTENT(IN)::UNICODESTR
  CHARACTER(LEN=*), INTENT(OUT)::MBSTR
  INTEGER(4), OPTIONAL, INTENT(IN)::FLAGS
  END FUNCTION
END INTERFACE
```

UNICODESTR    Input. INTEGER(2). Array of integers holding the Unicode string to be translated.

MBSTR                  Output. `CHARACTER(LEN=*)`. Translation of Unicode
                       string into multibyte character string from the current
                       codepage.

FLAGS                  Optional, input. `INTEGER(4)`. If specified, argument to
                       modify the string conversion. If `FLAGS` is omitted, no
                       extra checking of the conversion takes place. Available
                       values are:

                       `NLS$CompositeCheck`
                                             Convert composite characters to
                                             precomposed.

                       `NLS$SepChars`        Generate separate characters.

                       `NLS$DiscardDns`      Discard nonspacing characters.

                       `NLS$DefaultChars`
                                             Replace exceptions with default
                                             character.

The last three flags (`NLS$SepChars`, `NLS$DiscardDns`, and
`NLS$DefaultChars`) are mutually exclusive and can be used only if
`NLS$CompositeCheck` is set, in which case one (and only one) of them is
combined with `NLS$CompositeCheck` using an inclusive OR (`IOR` or
`OR`). These flags determine what translation to make when there is no
precomposed mapping for a base character/nonspace character combination
in the Unicode wide character string. The default
(`IOR(NLS$CompositeCheck, NLS$SepChars)`) is to generate
separate characters.

## Output

If no error occurs, returns the number of bytes written to `MBSTR` (bytes
are counted, not characters), or the number of bytes required to hold the
output string if `MBSTR` has zero length. If `MBSTR` is longer than the
translation, it is blank-padded. If `MBSTR` is zero length, the function
returns the number of bytes required to hold the translation and nothing
is written to `MBSTR`.

If an error occurs, one of the following negative values is returned:

`NLS$ErrorInsufficentBuffer`

> The `MBSTR` argument is too small, but not zero length so that the needed number of bytes is returned.

`NLS$ErrorInvalidFlags`

> The `FLAGS` argument has an illegal value.

# MBCS Fortran Equivalent Procedures

# MBINCHARQQ

*Same as INCHARQQ except that it can*
*read a single multi-byte character at once*
*and returns the number of bytes read*

## Prototype

```
INTERFACE
  INTEGER(4) FUNCTION MBInCharQQ(STRING)
  CHARACTER(LEN=2), INTENT(OUT)::STRING
                              ! LEN=MBLENMAX
  END FUNCTION
END INTERFACE
```

STRING      Output. `CHARACTER(MBLenMax)`. String containing the read characters, padded with blanks up to the length `MBLenMax`. The `MBLenMax` parameter, defined in the module `iflport.F90`, is the longest length, in bytes, of any character in any codepage installed on the system.

### Description

Performs the same function as INCHARQQ except that it can read a single multibyte character at once, and it returns the number of bytes read as well as the character.

### Output

Number of characters read.

# MBINDEX

*Same as INDEX, except that multi-byte characters can be included in its arguments*

### Prototype

```
INTERFACE
  INTEGER(4) FUNCTION MBIndex(STRING, SUBSTRING, &
                                   BACK)
  CHARACTER(LEN=*), INTENT(IN) :: STRING, SUBSTRING
  LOGICAL(4), INTENT(IN), OPTIONAL :: BACK
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| STRING | Input. CHARACTER(LEN=*). String to be searched for the presence of SUBSTRING. Can contain multibyte characters. |
| SUBSTRING | Input. CHARACTER(LEN=*). Substring whose position within STRING is to be determined. Can contain multibyte characters. |
| BACK | Optional, input. LOGICAL(4). If specified, determines direction of the search. If BACK is .FALSE. or is omitted, the search starts at the |

beginning of `STRING` and moves toward the end. If `BACK` is `.TRUE.`, the search starts end of `STRING` and moves toward the beginning.

### Description

Performs the same function as `INDEX` except that the strings manipulated can contain multibyte characters.

### Output

If `BACK` is omitted or is `.FALSE.`, returns the leftmost position in `STRING` that contains the start of `SUBSTRING`. If `BACK` is `.TRUE.`, returns the rightmost position in `STRING` which contains the start of `SUBSTRING`. If `STRING` does not contain `SUBSTRING`, returns 0. If `SUBSTRING` occurs more than once, returns the starting position of the first occurrence ("first" is determined by the presence and value of `BACK`).

# MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE

*Same as LGE, LGT, LLE, and LLT,*
*and the logical operators .EQ. and*
*.NE*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION MBLGE(STRA, STRB, FLAGS)
  CHARACTER(LEN=*), INTENT(IN) :: STRA, STRB
  INTEGER(4), INTENT(IN), OPTIONAL :: FLAGS
  END FUNCTION
END INTERFACE
```

STRA, STRB     Input. `CHARACTER(LEN=*)`. Strings to be compared. Can contain multibyte characters.

FLAGS        Optional, input. `INTEGER(4)`. If specified,
             determines which character traits to use or ignore
             when comparing strings. You can combine several
             flags using an inclusive `OR` (`IOR` or `OR`). There are no
             illegal combinations of flags, and the functions may
             be used without flags, in which case all flag options
             are turned off. The available values are:

`NLS$MB_IgnoreCase`
            Ignore case.

`NLS$MB_IgnoreNonspace`
            Ignore nonspacing characters (this flag
            removes Japanese accent characters if they
            exist).

`NLS$MB_IgnoreSymbols`
            Ignore symbols.

`NLS$MB_IgnoreKanaType`
              Do not differentiate between Japanese
              Hiragana and Katakana characters
              (corresponding Hiragana and Katakana
              characters will compare as equal).

`NLS$MB_IgnoreWidth`
            Do not differentiate between a single-byte
            character and the same character as a double
            byte.

`NLS$MB_StringSort`
            Sort all symbols at the beginning, including
            the apostrophe and hyphen (See **NOTE** that
            follows).

### Description

Perform the same functions as `LGE`, `LGT`, `LLE`, `LLT` and the logical operators `.EQ.` and `.NE.` except that the strings being compared can include multi-byte characters, and optional flags can modify the comparison. All these routines have the same arguments as shown for `MBLGE` above.

> **NOTE.** *If the strings supplied contain Arabic Kashidas, the Kashidas are ignored during the comparison. Therefore, if the two strings are identical except for Kashidas within the strings, the functions return a value indicating they are "equal" in the collation sense, though not necessarily identical.*

> **NOTE.** *When not using the* `NLS$MB_StringSort` *flag, the hyphen and apostrophe are special symbols and are treated differently than others. This is to ensure that words like coop and co-op stay together within a list. All symbols, except the hyphen and apostrophe, sort before any other alphanumeric character. If you specify the* `NLS$MB_StringSort` *flag, hyphen and apostrophe sort at the beginning also.*

### Output

Comparisons are made using the current locale, not the current codepage. The codepage used is the default for the language/country combination of the current locale.

The outputs of these functions are as follows:

- `MBLGE` returns `.TRUE.` if the strings are equal or `STRA` comes last in the collating sequence. Otherwise, it returns `.FALSE.`.
- `MBLGT` returns `.TRUE.` if `STRA` comes last in the collating sequence. Otherwise, it returns `.FALSE.`.

- MBLLE returns `.TRUE.` if the strings are equal or `STRA` comes first in the collating sequence. Otherwise, it returns `.FALSE.`.
- MBLLT returns `.TRUE.` if `STRA` comes first in the collating sequence. Otherwise, it returns `.FALSE.`.
- MBLEQ returns `.TRUE.` if the strings are equal in the collating sequence. Otherwise, it returns `.FALSE.`.
- MBLNE returns `.TRUE.` if the strings are not equal in the collating sequence. Otherwise, it returns `.FALSE.`.
- If the two strings are of different lengths, they are compared up to the length of the shortest one. If they are equal to that point, then the return value indicates that the longer string is greater.
- If `FLAGS` is invalid, the functions return `.FALSE.`.

# MBSCAN

*Same as SCAN, except that multi-byte characters can be included in its arguments*

### Prototype

```
INTERFACE
 INTEGER(4) FUNCTION MBScan(STRING, SET, BACK)
 CHARACTER(LEN=*), INTENT(IN)::STRING, SET
 LOGICAL(4), INTENT(IN), OPTIONAL::BACK
 END FUNCTION
END INTERFACE
```

STRING          Input. `CHARACTER(LEN=*)`. String to be searched for the presence of any character in `SET`.

SET             Input. `CHARACTER(LEN=*)`. Characters to search for.

BACK            Optional, input. `LOGICAL(4)`. If specified, determines direction of the search. If `BACK` is `.FALSE.` or is omitted, the search starts at the

beginning of STRING and moves toward the end. If BACK is .TRUE., the search starts end of STRING and moves toward the beginning.

### Description

Performs the same function as SCAN except that the strings manipulated can contain multibyte characters.

### Output

If BACK is .FALSE. or is omitted, returns the position of the leftmost character in STRING that is in SET. If BACK is .TRUE., returns the rightmost character in STRING that is in SET. If no characters in STRING are in SET, returns 0.

## MBVERIFY

*Same as VERIFY, except that multi-byte characters can be included in its arguments*

### Prototype

```
INTERFACE
    INTEGER(4) FUNCTION MBVerify(STRING, SET, BACK)
    CHARACTER(LEN=*), INTENT(IN)::STRING, SET
    LOGICAL(4), INTENT(IN), OPTIONAL::BACK
    END FUNCTION
END INTERFACE
```

| | |
|---|---|
| STRING | Input. CHARACTER(LEN=*). String to be searched for presence of any character not in SET. |
| SET | Input. CHARACTER(LEN=*). Set of characters tested to verify that it includes all the characters in string. |

BACK                Optional, input. `LOGICAL(4)`. If specified,
                    determines direction of the search. If `BACK` is
                    `.FALSE.` or is omitted, the search starts at the
                    beginning of `STRING` and moves toward the end. If
                    `BACK` is `.TRUE.`, the search starts end of `STRING` and
                    moves toward the beginning.

### Description

Performs the same function as `VERIFY` except that the strings manipulated
can contain multibyte characters.

### Output

If `BACK` is `.FALSE.` or is omitted, returns the position of the leftmost
character in `STRING` that is not in `SET`. If `BACK` is `.TRUE.`, returns the
rightmost character in `STRING` that is not in `SET`. If all the characters in
`STRING` are in `SET`, returns 0.

# MBJISTToJMS

*Converts a Japan Industry Standard
(JIS) character to a Microsoft Kanji
(Shift JIS or JMS) character*

### Prototype

```
INTERFACE
  CHARACTER(LEN=2) FUNCTION MBJISToJMS(CHAR)
  CHARACTER(LEN=2), INTENT(IN) :: CHAR
  END FUNCTION
END INTERFACE
```

CHAR                Input. `CHARACTER(LEN=2)`. JIS character to be
                    converted.

### Description

Converts a Japan Industry Standard (JIS) character to a Microsoft Kanji (Shift JIS or JMS) character.

A JIS character is converted only if the lead and trail bytes are in the hexadecimal range 21 through 7E.

---

**NOTE.** *Only computers with Japanese installed as one of the available languages can use the* MBJISToJMS *conversion function.*

---

### Output

MBJISToJMS returns a Microsoft Kanji (Shift JIS or JMS) character.

# MBJMSTToJIS

*Converts a Microsoft Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character*

### Prototype

```
INTERFACE
  CHARACTER(LEN=2) FUNCTION MBJMSToJIS( CHAR)
  CHARACTER(LEN=2), INTENT(IN) :: CHAR
  END FUNCTION
END INTERFACE
```

CHAR            Input. CHARACTER(LEN=2). JMS character to be converted.

### Description

Converts a Microsoft Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character.

A JMS character is converted only if the lead byte is in the hexadecimal range 81 through 9F or E0 through FC, and the trail byte is in the hexadecimal range 40 through 7E or 80 through FC.

**NOTE.** *Only computers with Japanese installed as one of the available languages can use the* MBJMSToJIS *conversion function.*

## Output

MBJMSToJIS returns a Japan Industry Standard (JIS) character.

# *POSIX Functions*   3

This chapter describes the functions that comprise the `POSIX` library
(`libPOSF90.lib`). These functions are made available to the compiler
when you invoke the `/4Yposixlib` option. These functions implement
the IEEE POSIX FORTRAN-77 Language bindings, as specified in IEEE
Standard 1003.9-1992. The POSIX standard is ISO/IEC 9945-1:1990.
Copies of the standard are available from IEEE.

The prototypes are described using the `INTERFACE` call, which provides
the required information to complete a call to the specified procedure. For
descriptions of the `INTERFACE` block and the `/4Yposixlib` option, see
the *Intel® Fortran Compiler User's Guide.*

## POSIX Library Interface

Depending on whether you are using free- ot fixed-form source, you can
interface to POSIX library in the following ways:

- With free-form source, to interface to `libPOSF90.lib` (`POSIX`
  library), your code should contain the

      INCLUDE 'iflposix.f90'

  statement  where `iflposix.f90` is the interface file.

**L**

- For fixed-form source, the `iflposix.f90` file has to be edited and compiled with the `/4L132` option. Or you can create a module like this:

```
MODULE iflposix
INCLUDE 'iflposix.f90'
END MODULE
```

In addition, in your fixed-form source include `USE iflposix` statement.

# PXFACCESS

*Determines the accessibility of the file*

### Prototype

```
INTERFACE
  SUBROUTINE PXFACCESS (PATH, ILEN, IAMODE,
        IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, IAMODE, IERROR
  END SUBROUTINE PXFACCESS
END INTERFACE
```

| | |
|---|---|
| PATH | name of the file |
| ILEN | length of PATH string |
| IAMODE | one of the following for Win32 systems: |
| | 0 -- checks for existence of the file |
| | 2 -- checks for write access |
| | 4 -- checks for read access |
| | 6 -- checks for read/write access |
| IERROR | return value |

### Description

Checks for the accessibility of a file or directory. On Win32 systems, if the name given is a directory name, then the function only checks for existence. All directories have read/write access on Win32 systems.

### Output

If the file does not exist, or the appropriate access is not available, an error code is returned in IERROR. Possible error codes include:

| | |
|---|---|
| -1 | a bad parameter was passed |
| EACCES | access requested was denied |
| ENOENT | file did not exist |

See your Microsoft Visual C++ installation in the include directory under errno.h for the values of EACCES and ENOENT.

# PXFAINTGET

*Gets an integer array component of a structure*

### Prototype

```
INTERFACE
   SUBROUTINE PXFAINTGET (JHANDLE, COMPNAME, VALUE,
        IALEN, IERROR)
   INTEGER(4) JHANDLE, IALEN, IERROR
   CHARACTER(LEN=*) COMPNAME
   INTEGER(4) VALUE(IALEN)
   END SUBROUTINE PXFAINTGET
END INTERFACE
```

| | |
|---|---|
| JHANDLE | handle to the structure |
| COMPNAME | component name |
| VALUE | (out) store the value of the component here |

| | |
|---|---|
| IALEN | length of the VALUE array |
| IERROR | return value |

### Description

This subroutine gets an integer array component of a structure.

### Output

If successful, IERROR is set to zero. ENONAME is set when no such component name exist for the structure. EARRAYLEN contains the number of array elements that exceeds IALEN.

# PXFAINTSET

*gets an integer array component of a structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFAINSET(JHANDLE, COMPNAME, VALUE,
      IALEN, IERROR)
  INTEGER(4) JHANDLE, IALEN, IERROR
  CHARACTER(LEN=*) COMPNAME
  INTEGER(4) VALUE(IALEN)
  END SUBROUTINE PXFAINTSET
END INTERFACE
```

| | |
|---|---|
| JHANDLE | handle to structure |
| COMPNAME | component name |
| VALUE | values to set |
| IALEN | length of the VALUE array |
| IERROR | return value |

### Description

This subroutine gets an integer array component of a structure.

### Output

If successful, IERROR is set to zero. Otherwise, ENONAME is set when no such component name is found in the structure. EARRAYLEN is set to the number of array elements that exceeds IALEN.

# PXFCALLSUBHANDLE

*Calls the associated subroutine*

### Prototype

```
INTERFACE
  SUBROUTINE PXFCALLSUBHANDLE(JHANDLE2, IVAL,
      IERROR)
  INTEGER(4) JHANDLE2, IVAL, IERROR
  END SUBROUTINE PXFCALLSUBHANDLE
END INTERFACE
```

| | |
|---|---|
| JHANDLE2 | handle to subroutine |
| IVAL | argument to subroutine |
| IERROR | (out) error status |

### Description

This subroutine, given a subroutine handle, calls the associated subroutine.

**NOTE.** *The subroutine shall not be a function, an intrinsic, or an entry point and shall be defined with exactly ONE integer argument.*

**Output**

If successful, IERROR is set to zero.

# PXFCHDIR

*Changes the current working directory*

**Prototype**

```
INTERFACE
  SUBROUTINE CHDIR(PATH, ILEN, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, IERROR
  END SUBROUTINE CHDIR
END INTERFACE
```

PATH            the directory to be changed to

ILEN            length of the PATH string

IERROR          return value

**Description**

This subroutine changes the current working directory.

**Output**

If successful, IERROR is set to zero.

# PXFCHMOD

*Changes the ownership mode of the file*

### Prototype

```
INTERFACE
  SUBROUTINE PXFCHMOD (PATH, ILEN, IMODE, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, IMODE, IERROR
  END SUBROUTINE PXFCHMOD
END INTERFACE
```

PATH            name of the file

ILEN            length of the PATH string

IMODE           mode

IERROR          return value

### Description

This subroutine changes the ownership mode of a file.

**NOTE.** On UNIX systems, you must have sufficient ownership permissions, such as being the owner of the file or having read/write access of the file

### Output

If successful, IERROR is set to zero.

---

# PXFCHOWN

*Changes the owner and group of a file*

---

### Prototype

```
INTERFACE
   SUBROUTINE PXFCHOWN(PATH, ILEN, IOWNER, IGROUP,
      IERROR)
   CHARACTER(LEN=*) PATH
   INTEGER(4) ILEN, IOWNER, IGROUP, IERROR
   END SUBROUTINE PXFCHOWN
END INTERFACE
```

| | |
|---|---|
| PATH | file or directory name |
| ILEN | length of the PATH string |
| IOWNER | owner uid |
| IGROUP | group gid |
| IERROR | (out) error status |

### Description

This subroutine changes the owner and group of a file.

### Output

If successful, IERROR is set to zero.

# PXFCLOSE

*Deletes a descriptor*

### Prototype

```
INTERFACE
  SUBROUTINE PXFCLOSE (FD, IERROR)
  INTEGER(4) FD, IERROR
  END SUBROUTINE PXFCLOSE
END INTERFACE
```

FD                file descriptor to be deleted

IERROR           (output) returned error code

### Description

This subroutine deletes a file descriptor.

### Output

If successful, IERROR is set to zero.

# PXFCLOSEDIR

*Closes the directory stream*

### Prototype

```
INTERFACE
  SUBROUTINE PXFCLOSEDIR (IDIRID, IERROR)
  INTEGER(4) IDIRID, IERROR
  END SUBROUTINE PXFCLOSEDIR
END INTERFACE
```

IDIRID           pointer to DIR structure

IERROR           (out) error status

### Description

This subroutine closes the directory stream and frees the DIR structure.

### Output

If successful, IERROR is set to zero.

# PXFCONST

*Retries the value associated with a constant*

### Prototype

```
INTERFACE
  SUBROUTINE PXFCONST (CONSTNAME, IVAL, IERROR)
  CHARACTER(LEN=*) CONSTNAME
  INTEGER(4) IVAL, IERROR
  END SUBROUTINE PXFCONST
END INTERFACE
```

| | |
|---|---|
| CONSTNAME | name of the constant |
| IVAL | (out) value of the constant |
| IERROR | (out) return value |

### Description

This subroutine retrieves the value associated with a constant name. you can inquire about the values of the following symbolic constants:

STDIN_UNIT
STDOUT_UNIT
STDERR_UNIT
EINVAL
ENONAME
ENOHANDLE
EARRAYLEN

The constants beginning with `E` signify various error values for the system variable `errno`. Check your MSVC++ documentation for more information, on Win32 systems. On UNIX, look at the `/usr/include/errno.h` file.

### Output

If successful, `IERROR` is set to zero; otherwise, it is set to `-1`.

# PXFCREAT

*Creates a new file or rewrites an existing one*

### Prototype

```
INTERFACE
   SUBROUTINE PXFCREAT (PATH, ILEN, IMODE, IFILDES,
           IERROR)
   CHARACTER(LEN=*) PATH
   INTEGER(4) ILEN, IMODE, IFILDES, IERROR
   END SUBROUTINE PFXCREAT
END INTERFACE
```

| | |
|---|---|
| PATH | pathname of the file |
| ILEN | length of PATH string |
| IMODE | mode of the newly created file |
| IFILDES | (output) file descriptor |
| IERROR | return value |

### Description

This subroutine creates a new file or rewrites an existing one.

**Output**

If successful, IERROR is set to zero.

# PXFDUP

*Duplicates an existing file descriptor*

### Prototype

```
INTERFACE
  SUBROUTINE PXFDUP (IFILDES, IFID, IERROR)
  INTEGER(4) IFILDES, IFID, IERROR
  END SUBROUTINE PXFDUP
END INTERFACE
```

| | |
|---|---|
| IFILDES | descriptor to be duplicated |
| IFID | (output) handle for the duplicated descriptor |
| IERROR | (output) returned error code |

### Description

This subroutine duplicates an existing file descriptor.

### Output

If successful, IERROR is set to zero.

# PXFDUP2

*Duplicates an existing file descriptor*

### Prototype

```
INTERFACE
  SUBROUTINE PXFDUP2 (IFILDES, IFILDES2, IERROR)
  INTEGER(4) IFILDES, IFILDES2, IERROR
  END SUBROUTINE PXFDUP2
END INTERFACE
```

IFILDES           descriptor to be duplicated

IFILDES2          desired handle for the duplicated descriptor

IERROR            (output) returned error code

### Description

This subroutine duplicates an existing file descriptor.

### Output

If successful, IERROR is set to zero.

# PXFEINTGET

*Gets a single element from an integer*
*array component of a structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFEINTGET (JHANDLE, COMPNAME, INDEX,
    VALUE, IERROR)
  CHARACTER(LEN=*) COMPNAME
  INTEGER(4) JHANDLE, INDEX, VALUE, IERROR
  END SUBROUTINE PXFEINTGET
END INTERFACE
```

| | |
|---|---|
| JHANDLE | handle to structure |
| COMPNAME | component name |
| INDEX | index of element |
| VALUE | (out) where the value of the component is stored |
| IERROR | return value |

### Description

This subroutine gets a single element from an integer array component of a structure.

### Output

If successful, IERROR is set to zero. Otherwise, ENONAME is set when EINVAL is set if an invalid index is specified.

# PXFEINTSET

*Sets a single element from an integer
array component of a structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFEINTSET (JHANDLE, COMPNAME,
    IVALUE, INDEX, IERROR)
  CHARACTER(LEN=*) COMPNAME
  INTEGER(4) JHANDLE, IVALUE, INDEX, IERROR
  END SUBROUTINE PXFEINTSET
END INTERFACE
```

| | |
|---|---|
| JHANDLE | handle to structure |
| COMPNAME | component name |
| IVALUE | index of element |
| INDEX | value to set |
| IERROR | return value |

### Description

This subroutine sets a single element from an integer array component of a
structure.

### Output

If successful, IERROR is set to zero. Otherwise, ENONAME is set when no
such component name exist for the structure. EINVAL is set when an
invalid index is specified.

# PXFESTRGET

*Gets a single element from a string
array component of a structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFESTRGET (JHANDLE, COMPNAME, INDEX,
   VALUE, ILEN, IERROR)
  INTEGER(4) JHANDLE, INDEX, ILEN, IERROR
  CHARACTER(LEN=*) COMPNAME, VALUE
  END SUBROUTINE PXFESTRGET
END INTERFACE
```

| | |
|---|---|
| JHANDLE | handle to structure |
| COMPNAME | component name |
| INDEX | index of element |
| VALUE | (out) where the string is stored |
| ILEN | (out) length of string |
| IERROR | return value |

### Description

This subroutine gets a single element from a string array component of a
structure.

### Output

If successful, IERROR is set to zero. Otherwise, ENONAME is set when no
such component name exist for the structure. EINVAL is set when an
invalid index is specified.

# PXFEXECV

*Executes an executable*

### Prototype

```
INTERFACE
   SUBROUTINE PXFEXECV (PATH, LENPATH, ARGV,
     LENARGV, IARGC, IERROR)
   INTEGER(4) LENPATH, LENGARV(0:IARGC-1), IARGC,
              IERROR
   CHARACTER(LEN=*) PATH, ARGV(0:IARGC-1)
   END SUBROUTINE PXFEXECV
END INTERFACE
```

| | |
|---|---|
| PATH | (input) new executable path |
| LENPATH | (input) length of PATH string |
| ARGV | (input) command-line arguments |
| LENARGV | (input) length of each argument string |
| IARGC | (input) argument count |
| IERROR | (output) error value |

### Description

This subroutine executes an executable file.

### Output

Zero if successful, otherwise, an error code, the value of errno.

# PXFEXECVE

*Executes an executable with*
*environment settings*

### Prototype

```
INTERFACE
  SUBROUTINE PXFEXECVE (PATH, LENPATH, ARGV,
    LENARGV, IARGC, ENV, LENENV, IENVC, IERROR)
  INTEGER(4) LENPATH, LENARGV(0:IARGC-1), IARGC,
    LENENV(IENVC), IENVC, IERROR
  CHARACTER(LEN=*) PATH, ARGV(0:IARGC-1), ENV(IENVC)
  END SUBROUTINE PXFEXECVE
END INTERFACE
```

| | |
|---|---|
| PATH | (input) new executable path |
| LENPATH | (input) length of PATH string |
| ARGV | (input) command-line arguments |
| LENARGV | (input) length of each argument string |
| IARGC | (input) argument count |
| ENV | (input) environment setting |
| LENENV | (input) length of elements in ENV |
| IENVC | (input) number of element sin ENV |
| IERROR | return value |

### Description

This subroutine executes an executable file with environment settings.

### Output

None.

# PXFEXECVP

*Executes a file*

### Prototype

```
INTERFACE
  SUBROUTINE PXFEXECVP (FILE, LENFILE, ARGV,
    LENARGV, IARGC, IERROR)
  INTEGER(4) LENFILE, LENARGV(0:IARGC-1), IARGC,
             IERROR
  CHARACTER(LEN=*) FILE, ARGV(0:IARGC-1)
  END SUBROUTINE PXFEXECVP
END INTERFACE
```

| | |
|---|---|
| FILE | (input) file to be executed |
| LENFILE | (input) length of FILE |
| ARGV | (input) command-line arguments |
| LENARGV | (input) length of each argument string |
| IARGC | (input) argument count |
| IERROR | (output) error value |

### Description

Executes a file.

### Output

None.

---

# PXFEXIT

*Exits from a process*

---

### Prototype

```
INTERFACE
  SUBROUTINE PXFEXIT (ISTATUS)
  INTEGER(4) ISTATUS
  END SUBROUTINE PXFEXIT
END INTERFACE
```

ISTATUS          (input) exit value

### Description

This subroutine exits a process.

### Output

None.

---

# PXFFASTEXIT

*Exits from the process*

---

### Prototype

```
INTERFACE
  SUBROUTINE PXFFASTEXIT (ISTATUS)
  INTEGER(4) ISTATUS
  END SUBROUTINE PXFFASTEXIT
END INTERFACE
```
ISTATUS          (input) exit value

---

### Description

This subroutine exits from the process. There is no possible return from this subroutine.

### Output

None.

# FFLUSH

*Writes any unwritten data for an output*
*stream*

### Prototype

```
INTERFACE
  SUBROUTINE FFLUSH (LUNIT, IERROR)
  INTEGER(4) IUNIT, IERROR
  END SUBROUTINE FFLUSH
END INTERFACE
```

| | |
|---|---|
| LUNIT | a FORTRAN logical unit, which must be an `INTEGER*4` expression with the value in the range of 0 to 100. |
| IERROR | return value |

### Description

This subroutine writes any unwritten data for an output stream.

### Output

If successful, `IERROR` is set to zero. Otherwise, `EINVAL` is set if no file connected to `LUNIT`. `EFBIG` is set if the file exceeds maximum file size. `ENOSPC` is set if no free space is available on the device. `ESPIPE` is set if `LUNIT` is a pipe or FIFO.

# FGETC

*Reads a character from the specified
stream (logical unit)*

### Prototype

```
INTERFACE
  SUBROUTINE FGETC (LUNIT, CHAR, IERROR)
  CHARACTER(LEN=1) CHAR
  INTEGER(4) IERROR
  END SUBROUTINE FGETC
END INTERFACE
```

| | |
|---|---|
| LUNIT | a FORTRAN logical unit, which must be an INTEGER*4 expression with the value in the range of 0 to 100 |
| CHAR | character to be read |
| IERROR | return value |

### Description

This subroutine reads a character from the specified stream (logical unit).

### Output

If successful, IERROR is set to zero.

# PXFFILENO

*Returns the file descriptor associated with a stream*

### Prototype

```
INTERFACE
  SUBROUTINE PXFFILENO (LUNIT, FD, IERROR)
  INTEGER(4) LUNIT, FD, IERROR
  END SUBROUTINE PXFFILENO
END INTERFACE
```

| | |
|---|---|
| LUNIT | a FORTRAN logical unit, which must be an INTEGER*4 expression with the value in the range of 0 to 100 |
| FD | (output) file descriptor |
| IERROR | (output) returned error code |

### Description

This subroutine returns the file descriptor associated with a stream.

### Output

If successful, IERROR is set to zero. Otherwise, EINVAL is set if LUNIT is not an open unit. EBADF is set if LUNIT is not connected with a file descriptor.

# PXFFORK

*Fork a child process*

### Prototype

```
INTERFACE
  SUBROUTINE  PXFFORK (IPID, IERROR)
  INTEGER(4) IPID, IERROR
  END SUBROUTINE PXFFORK
END INTERFACE
```

| | |
|---|---|
| IPID | (output) process ID |
| IERROR | (output) Zero if the process is started successfully. Otherwise contains an appropriate error value of ERRNO. |

### Description

This subroutine spawns an asynchronous child process.

### Output

A process handle or process ID is returned in the variable IPID if the process was successfully forked. Otherwise, the returned IPID value is -1.

# FPUTC

*Outputs a character to the specified stream (logical unit)*

### Prototype

```
INTERFACE
  SUBROUTINE FPUTC (LUNIT, CHAR, IERROR)
  CHARACTER(LEN=1) CHAR
  INTEGER(4) LUNIT, IERROR
```

```
      END SUBROUTINE FPUTC
END INTERFACE
```

| | |
|---|---|
| LUNIT | a FORTRAN logical unit, which must be an `INTEGER*4` expression with the value in the range of 0 to 100 |
| CHAR | character to be written |
| IERROR | return value |

### Description

This subroutine outputs a character to the specified stream.

### Output

If successful, `IERROR` is set to zero. `EEND` is set if end of file is reached.

# FSEEK

*Sets the position of the next input / output on the stream*

### Prototype

```
INTERFACE
   SUBROUTINE FSEEK (LUNIT, IOFFSET, IWHENCE,
       IERROR)
   INTEGER(4) LUNIT, IOFFSET, IWHENCE, IERROR
   END SUBROUTINE FSEEK
END INTERFACE
```

| | |
|---|---|
| LUNIT | a FORTRAN logical unit, which must be an `INTEGER*4` expression with the value in the range of 0 to 100. |
| IOFFSET | number of bytes away from whence to place the pointer |
| IWHENCE | position within the stream |

IERROR          return value

## Description

This subroutine sets the position of the next input/output on the stream.

## Output

If successful, IERROR is set to 0. EINVAL is set if no file connected to LUNIT if IWHENCE is not a proper value or resulting offset would be invalid. ESPIPE is set if LUNIT is a pipe or FIFO. EEND is set if end of file is reached.

# PXFFSTAT

*Gets a file status*

## Prototype

```
INTERFACE
  SUBROUTINE PXFFSTAT (IFILDES, JSTAT, IERROR)
  INTEGER(4) IFILDES, JSTAT, IERROR
  END SUBROUTINE PXFFSTAT
END INTERFACE
```

IFILDES       file handle

JSTAT         pointer to the PXFFSTAT structure

IERROR        return value

## Description

This subroutine gets the status of a file.

## Output

If successful, IERROR is set to zero.

# FTELL

*Returns the relative position in bytes
from the beginning of the file*

### Prototype

```
INTERFACE
  SUBROUTINE FTELL (LUNIT, IOFFSET, IERROR)
  INTEGER(4) LUNIT, IOFFSET, IERROR
  END SUBROUTINE FTELL
END INTERFACE
```

| | |
|---|---|
| LUNIT | a FORTRAN logical unit, which must be an INTEGER*4 expression with the value in the range of 0 to 100 |
| IOFFSET | (output) relative position in bytes from beginning of the file |
| IERROR | return value |

### Description

This subroutine returns the relative position in bytes from the beginning of the file.

### Output

If successful, IERROR is set to zero. EINVAL is set if no file connected to LUNIT. ESPIPE is set if LUNIT is connected to a pipe or FIFO.

---

# GETC

*Reads a character from standard input, unit 5*

---

### Prototype

```
INTERFACE
   SUBROUTINE GETC (NEXTCHAR, IERROR)
   CHARACTER(LEN=1) NEXTCHAR
   INTEGER(4) IERROR
   END SUBROUTINE GETC
END INTERFACE
```

NEXTCHAR        character to be read

IERROR          return value

### Description

This subroutine reads a character from the standard input, unit 5.

### Output

If successful, IERROR is set to zero.

# GETCWD

*Retrieves the path of the current
working directory*

### Prototype

```
INTERFACE
  SUBROUTINE GETCWD (BUF, ILEN, IERROR)
  CHARACTER(LEN=*) BUF
  INTEGER(4) ILEN, IERROR
  END SUBROUTINE GETCWD
END INTERFACE
```

| | |
|---|---|
| BUF | (output) storage for the retrieved pathname |
| ILEN | (output) length of the retrieved pathname |
| IERROR | return value |

### Description

This subroutine retrieves the path of the current working directory.

### Output

If successful, IERROR is set to zero. EINVAL is set if the size of BUF is
insufficient.

# PXFGETGRGID

*Gets entry with specified GID*

### Prototype

```
INTERFACE
  SUBROUTINE PXFGETGRGID (JGID, JGROUP, IERROR)
  INTEGER(4) IGID, JGROUP, IERROR
  END SUBROUTINE PXFGETGRGID
END INTERFACE
```

| | |
|---|---|
| JGID | group ID |
| JGROUP | (output) pointer to the structure PXFGROUP |
| IERROR | (output) error status |

### Description

This subroutine gets entry with specified group ID.

### Output

If successful, IERROR is set to zero.

# PXFGETGRNAM

*Get entry with the specified group name*

### Prototype

```
INTERFACE
  SUBROUTINE PXFGETGRNAM (NAME, ILEN, JGROUP,
          IERROR)
  CHARACTER(LEN=*) NAME
  INTEGER(4) ILEN, JGROUP, IERROR
  END SUBROUTINE PXFGETGRNAM
END INTERFACE
```

| | |
|---|---|
| NAME | name of group |
| ILEN | length of NAME string |
| JGROUP | (output) pointer to the structure PXFGROUP |
| IERROR | (output) error status |

### Description

This subroutine gets entry with the specified group name.

### Output

If successful, IERROR is set to zero.

# PXFGETPWNAM

*Gets entry with specified user name*

### Prototype

```
INTERFACE
  SUBROUTINE PXFGETPWNAM (NAME, ILEN, JPASSWD,
     IERROR)
  CHARACTER(LEN=*) NAME
  INTEGER(4) JPASSWD, ILEN, IERROR
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| NAME | login name of the user |
| ILEN | length of the NAME string |
| JPASSWD | (output) pointer to the structure PXFPASSWD |
| IERROR | (output) error status |

### Description

This subroutine gets entry with specified user name. For example, a login name might be "jsmith" while the actual name is "John Smith."

### Output

If successful, `IERROR` is set to zero.

# PXFGETPWUID

*Gets entry with specified UID*

### Prototype

```
INTERFACE
  SUBROUTINE PXFGETPWUID (IUID, JPASSWD, IERROR)
  INTEGER(4) IUID, JPASSWD, IERROR
  END SUBROUTINE PXFGETPWUID
END INTERFACE
```

IUID           user ID

JPASSWD        (output) pointer to the structure `PXFPASSWD`

IERROR         (output) error status

### Description

This subroutine gets entry with specified user ID.

### Output

If successful, `IERROR` is set to zero.

# PXFGETSUBHANDLE

*Returns a subroutine handle for a
specified subroutine*

### Prototype

```
INTERFACE
   SUBROUTINE PXFGETSUBHANDLE (SUB, JAHNDLE1,
                                IERROR)
   INTEGER(4) JHANDLE1, IERROR
   EXTERNAL SUB
   END SUBROUTINE PXFGETSUBHANDLE
END INTERFACE
```

SUB             pointer to a subroutine

JAHNDLE1        (output) handle to subroutine

IERROR          (output) error status

### Description

This subroutine returns a subroutine handle for a specified subroutine.

**NOTE.** *The argument* SUB *shall not be a function, an intrinsic, or an entry point, and shall be defined with exactly one integer argument.*

### Output

If successful, IERROR is set to zero.

# PXFINTGET

*Sets an integer component of a structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFINTGET (JHANDLE, COMPNAME,
    VALUE, IERROR)
  INTEGER(4) JHANDLE, VALUE, IERROR
  CHARACTER(LEN=*) COMPNAME
  END SUBROUTINE PXFINTGET
END INTERFACE
```

| | |
|---|---|
| JHANDLE | handle to the structure |
| COMPNAME | component name |
| VALUE | (output) where the value of the component is stored |
| IERROR | return value |

### Description

This subroutine sets an integer component of a structure.

### Output

If successful, IERROR is set to zero. ENONAME is set if no such component name for the structure.

# PXFINTSET

*Sets an integer component of a structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFINTSET (JHANDLE, COMPNAME,
    VALUE, IERROR)
  INTEGER(4) JHANDLE, IERROR, VALUE
  CHARACTER(LEN=*) COMPNAME
  END SUBROUTINE
END INTERFACE
```

JHANDLE          handle to the structure

COMPNAME       component name

VALUE             set to this value

IERROR           return value

### Description

This subroutine sets an integer component of a structure.

### Output

If successful, IERROR is set to zero. ENONAME is set if no such component
name for the structure.

# PXFISBLK

*Tests for block special file*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFISBLK (M)
  INTEGER(4) M
  END FUNCTION PXFISBLK
END INTERFACE
```

M               value of the ST_MODE component in the structure
                PXFSTAT (stat)

### Description

This function tests for block special file.

### Output

A logical value is returned showing whether the specified file handle
corresponds to a block device: .TRUE. if the file is a block mode file,
.FALSE. otherwise.

# PXFISCHR

*Checks if character file*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFISCHR (M)
  INTEGER(4) m
  END FUNCTION PXFISCHR
END INTERFACE
```

M                value of the ST_MODE component in the structure
                 PXFSTAT (stat)

### Description

This function checks if the file is character file.

### Output

A logical value is returned, showing whether the specified file handle
corresponds to a character mode device: .TRUE. if the logical unit M is
connected to a character mode device, .FALSE. otherwise.

## PXFISDIR

*Checks if directory*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFISDIR (M)
  INTEGER(4) M
  END FUNCTION PXFISDIR
END INTERFACE
```

M                value of the ST_MODE component in the structure
                 PXFSTAT (stat)

### Description

This subroutine checks if the component is a directory.

### Output

Returns .TRUE. if M is a file handle representing a directory, .FALSE.
otherwise.

## PXFISFIFO

*Checks to determine whether a file is a*
*special FIFO file*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFISFIFO (M)
  INTEGER(4) M
  END FUNCTION PXFISFIFO
END INTERFACE
```

M     value of the `ST_MODE` component in the structure
      `PXFSTAT` (stat)

### Description

This function checks to determine whether a file is a special FIFO file
created by `PXFMKFIFO`.

**NOTE.** *On Win32 systems, this function always returns an error code of*
*`-1`. Win32 does not support symbolic file links.*

### Output

A true/false value that indicates whether the file is a special FIFO file.

# PXFISREG

*Checks if a file is a special FIFO file.*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFISREG (M)
  INTEGER(4) M
  END FUNCTION PXFISREG
END INTERFACE
```

M     value of the `ST_MODE` component in the structure
      `PXFSTAT` (stat)

### Description

This function checks if a LOGICAL(4) value is true/false to determine if the queried file is a special FIFO file. If PXFISREG returns true, the file is a regular file.

### Output

A logical true/false value.

## PXFKILL

*Kills the specified process.*

### Prototype

```
INTERFACE
  SUBROUTINE PXFKILL (IPID, ISIG, IERROR)
  INTEGER(4) IPID, ISIG, IERROR
  END SUBROUTINE PXFKILL
END INTERFACE
```

| IPID | specifies what process to kill as determined by its value: | |
|---|---|---|
| | > 0 | the specific process is killed. |
| | < 0 | all processes in the group are killed. |
| | == 0 | all processes in the group except special processes are killed. |
| | == pid_t-1 | all processes are killed. |
| ISIG | value of desired signal | |
| IERROR | return value | |

### Description

This subroutine kills the specified process.

### Output

`IERROR` contains a nonzero if child process exited normally.

# PXFLINK

*Links to a file/directory*

### Prototype

```
INTERFACE
  SUBROUTINE PXFLINK (EXISTING, LENEXIST, NEW,
        LENNEW, IERROR)
  CHARACTER(LEN=*) EXISTING, NEW
  INTEGER(4) LENEXIST, LENNEW, IERROR
  END SUBROUTINE PXFLINK
END INTERFACE
```

| | |
|---|---|
| EXISTING | existing file/directory name |
| LENEXIST | length of `EXISTING` string |
| NEW | new file/directory name |
| LENNEW | length of `NEW` string |
| IERROR | error status |

### Description

This subroutine links to a file/directory.

### Output

If successful, `IERROR` is set to zero.

# PXFLOCALTIME

*Converts a given elapsed time in
seconds into current system date*

### Prototype

```
INTERFACE
  SUBROUTINE PXFLOCALTIME (STIME, DATEARRAY,
        IERROR)
  INTEGER(4) ISECNDS, DATEARRAY(9), IERROR
  END SUBROUTINE PXFLOCALTIME
END INTERFACE
```

| | |
|---|---|
| STIME | elapsed time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970. |
| DATEARRAY | will contain current date and system time: |

|  |  |
|---|---|
| (1) seconds | (0-59) |
| (2) minutes | (0-59) |
| (3) hours | (0-23) |
| (4) day of month | (1-31) |
| (5) month | (1-12) |
| (6) year | (Gregorian)(ex. 1990) |
| (7) day of week | (0-6, 0 is Sunday) |
| (8) day of year | (1-366) |
| (9) daylight savings | (1, if in effect; 0, otherwise) |

| | |
|---|---|
| IERROR | return value |

### Description

Converts a given elapsed time in seconds into current system date.

### Output

If successful, IERROR is set to zero. EINVAL is set if the current value of
TZ is invalid.

# PXFLSEEK

*This function positions a file a specified distance in bytes*

### Prototype

```
INTERFACE
  SUBROUTINE PXFLSEEK (IFILDES, OFFSET, WHENCE,
    POSITION, IERROR)
  INTEGER(4) IFILDES, OFFSET, WHENCE, POSITION,
IERROR
  END SUBROUTINE PXFLSEEK
END INTERFACE
```

IFILDES        file descriptor

OFFSET         number of bytes to move

WHENCE         starting position

POSITION       (output) ending position

IERROR         returned error code

### Description

This subroutine positions a file a specified distance (OFFSET) in bytes depending upon the input value of WHENCE. Where WHENCE is one of the following:

SEEK_SET 0

SEEK_CUR 1

SEEK_END 2

If WHENCE is SEEK_SET, the file is positioned to OFFSET bytes from the beginning of the file. If WHENCE is SEEK_CUR, the file is positioned to OFFSET bytes from the current position of the file. If WHENCE is SEEK_END the file is positioned OFFSET bytes beyond the end of the line.

**Output**

If successful, IERROR is set to zero.

# MKDIR

*Makes a directory*

**Prototype**

```
INTERFACE
  SUBROUTINE MKDIR (PATH, ILEN, IMODE, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, IMODE, IERROR
  END SUBROUTINE MKDIR
END INTERFACE
```

PATH            the directory to be changed to

ILEN            length of PATH string

IMODE           mode mask

IERROR          return value

**Description**

This subroutine creates a directory.

**Output**

If successful, IERROR is set to zero.

# PXFMKFIFO

*Creates a new FIFO*

### Prototype

```
INTERFACE
  SUBROUTINE PXFMKFIFO (PATH, ILEN, IMODE,
    IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, IMODE, IERROR
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| PATH | name of FIFO file to create |
| ILEN | length of PATH string |
| IMODE | file permission (bits) |
| IERROR | error status |

### Description

This subroutine creates a new FIFO.

### Output

If successful, IERROR is set to zero.

# PXFOPEN

*Opens/creates a file for reading or writing*

### Prototype

```
INTERFACE
  SUBROUTINE PXFOPEN (PATH, ILEN, IOPENFLAG,&
    IMODE, IFILDES, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, OPENFLAG, IMODE, IFILDES, IERROR
  END SUBROUTINE
END INTERFACE
```

PATH            pathname of the file

ILEN            length of PATH string

IOPENFLAG       specifies how to open the file (read/write)

IMODE           mode of the newly-created file, if
                    IOPENFLAG = O_CREAT

IFILDES         (output) file descriptor

IERROR          return value

### Description

This subroutines creates a file or opens an existing one for read/write.

### Output

If successful, IERROR is set to zero.

# PXFOPENDIR

*Opens a directory and associates a stream with it*

### Prototype

```
INTERFACE
  SUBROUTINE PXFOPENDIR (DIRNAME, LENDIRNAME,
OPENDIRID, IERROR)
  CHARACTER(LEN=*) DIRNAME
  INTEGER(4) LENDIRNAME, IOPENDIRID, IERROR
  END SUBROUTINE PXFOPENDIR
END INTERFACE
```

| | |
|---|---|
| DIRNAME | directory name |
| LENDIRNAME | length of DIRNAME string |
| OPENDIRID | (out) pointer to DIR structure |
| IERROR | (out) error status |

### Description

This subroutine opens a directory and associates a stream with it.

### Output

If successful, IERROR is set to zero.

# PXFPIPE

*Creates a communications pipe between*
*two processes*

### Prototype

```
INTERFACE
  SUBROUTINE PXFPIPE (IREADFD, IWRITEFD, IERROR)
  INTEGER(4) IREADFD, IWRITEFD, IERROR
  END SUBROUTINE PXFPIPE
END INTERFACE
```

| IREADFD | (output) reading file descriptor |
| IWRITEFD | (output) writing file descriptor |
| IERROR | (output) returned error code |

### Description

This subroutine creates a communications pipe between two processes.

### Output

If successful, IERROR is set to zero.

# PXFPUTC

*Outputs a character to logical unit 6*
*(stdout)*

### Prototype

```
INTERFACE
  SUBROUTINE PXFPUTC (CH, IERROR)
  CHARACTER(LEN=1) CH
  INTEGER(4) IERROR
  END SUBROUTINE PXFPUTC
END INTERFACE
```

CH              character to be written

IERROR          returned error code

### Description

This subroutine outputs a character to logical unit 6 (stdout).

### Output

If successful, IERROR is set to zero. EEND is set if end of file is encountered.

# PXFREAD

*Reads from a file*

### Prototype

```
INTERFACE
  SUBROUTINE PXFREAD (IFILDES, CHAR, NBYTE, NREAD,
  IERROR)
   INTEGER(4) IFILDES
   CHARACTER(LEN=*) BUF
   INTEGER(4) NBYTE, NREAD, IERROR
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| IFILDES | descriptor to be read from |
| CHAR | buffer to be written to |
| NBYTE | number of bytes to read |
| NREAD | (output) number of bytes read |
| IERROR | (output) returned error code |

### Description

This subroutine reads a specified number of bytes from a file.

### Output

If successful, IERROR is set to zero. Otherwise, NREAD is undefined if error occurs.

# PXFREADDIR

*Reads the current directory entry*

### Prototype

```
INTERFACE
  SUBROUTINE PXFREADDIR (IDIRID, JDIRENT, IERROR)
  INTEGER(4) IDIRID, JDIRENT, IERROR
  END SUBROUTINE PXFREADDIR
END INTERFACE
```

IDIRID        pointer to `DIR` structure

JDIRENT       (out) pointer to `PXFDIRENT` structure

IERROR        (out) error status

### Description

This subroutine reads the current directory entry.

### Output

If successful, `IERROR` is set to zero.

# PXFRENAME

*Changes the name of a file*

### Prototype

```
INTERFACE
  SUBROUTINE PXFRENAME (OLD, LENOLD, NEW, LENNEW,
   IERROR)
  CHARACTER(LEN=*) OLD, NEW
  INTEGER(4) LENOLD, LENNEW, IERROR
  END SUBROUTINE PXFRENAME
END INTERFACE
```

| | |
|---|---|
| OLD | old filename |
| LENOLD | length of OLD string |
| NEW | new filename |
| LENNEW | length of NEW string |
| IERROR | return value |

### Description

This subroutine changes the name of file.

### Output

If successful, IERROR is set to zero.

# PXFREWINDDIR

*Resets the position of the stream to the beginning of the directory*

### Prototype

```
INTERFACE
  SUBROUTINE PXFREWINDDIR (IDIRID, IERROR)
  INTEGER(4) IDIRID, IERROR
  END SUBROUTINE PXFREWINDDIR
END INTERFACE
```

| | |
|---|---|
| IDIRID | pointer to DIR structure |
| IERROR | (out) error status |

### Description

This subroutine resets the position of the stream to the beginning of the directory.

### Output

If successful, `IERROR` is set to zero.

# PXFRMDIR

*Removes a directory*

### Prototype

```
INTERFACE
  SUBROUTINE PXFRMDIR (PATH, ILEN, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, IERROR
  END SUBROUTINE PXFRMDIR
END INTERFACE
```

PATH            the directory to be removed

ILEN            length of PATH string

IERROR          return value

### Description

This subroutine removes a directory.

### Output

If successful, `IERROR` is set to zero.

# PXFSIGADDSET

*Adds a signal to the set*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSIGADDSET (JSIGNET, ISIGNO, IERROR)
  INTEGER(4) JSIGNSET, ISIGNO, IERROR
  END SUBROUTINE PXFSIGADDSET
END INTERFACE
```

| | |
|---|---|
| JSIGNET | a bit vector structure created via PXFSTRUCTURECREATE. |
| ISIGNO | (input) signal to be added to the set |
| IERROR | return code |

### Description

Adds a signal to the set JSIGNET.

**NOTE.** *You must use the* PXFSTRUCTURECREATE *subroutine to create the* JSIGNSET *structure.* JSIGNSET *is not inherited by child applications started as a result of* PXFFORK.

### Output

If successful, IERROR is set to zero.

# PXFSIGDELSET

*Deletes a signal*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSIGDELSET (JSIGSET, ISIGNO, IERROR)
  INTEGER(4) JSIGSET, ISIGNO, IERROR
  END SUBROUTINE PXFSIGDELSET
END INTERFACE
```

| | |
|---|---|
| JSIGNET | a set representing a group of signals (obtained with PXFSTRUCTURECREATE) |
| ISIGNO | (input) signal to be deleted to the set |
| IERROR | return code |

### Description

Deletes the value of a given signal, ISIGNO, from the set JSIGNSET. After the signal has been deleted, any occurrence of the signal will not be "caught" by the application, and will be re-signaled, causing possible fatal application errors.

### Output

If successful, IERROR is set to zero.

# PXFSIGEMPTYSET

*Determines whether a signal set is empty.*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSIGEMPTYSET (JSIGSET, IERROR)
  INTEGER(4) JSIGSET, IERROR
  END SUBROUTINE
END INTERFACE
```

JSIGNET      a set representing a group of signals (obtained with `PXFSTRUCTURECREATE`)

IERROR      return code

### Description

Tests to determine whether the set given by `JSIGNSET` is empty.

> **NOTE.** *You must use the* `PXFSTRUCTURECREATE` *subroutine to create the* `JSIGNSET` *structure.*

### Output

If successful, `IERROR` is set to zero; non-zero for a non-empty set.

# PXFSIGFILLSET

*Fill a signal set.*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSIGFILLSET (JSIGSET, IERROR)
  INTEGER(4) JSIGSET, IERROR
  END SUBROUTINE PXFSIGFILLSET
END INTERFACE
```

JSIGNET       a set representing a group of signals (obtained with
              PXFSTRUCTURECREATE)

IERROR        return code

### Description

This subroutine adds a signal to the JSIGNSET. It is useful for initializing a
set.

**NOTE.** *You must use the* PXFSTRUCTURECREATE *subroutine to create
the* JSIGNSET *structure.*

### Output

If successful, IERROR is set to zero.

# PXFSIGISMEMBER

*Checks if signal is a member of a set*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSIGISMEMBER (JSIGSET, ISIGNO,
ISMEMBER,
   IERROR)
  INTEGER(4) JSIGSET, ISIGNO, IERROR
  LOGICAL(4) ISMEMBER
  END SUBROUTINE PXFSIGISMEMBER
END INTERFACE
```

| | |
|---|---|
| JSIGNET | a set representing a group of signals (obtained with PXFSTRUCTURECREATE) |
| ISIGNO | (input) signal to be tested for membership |
| ISMEMBER | (output) logical result |
| IERROR | return code |

### Description

This subroutine checks if the specified signal is a member of the specified set.

### Output

If true, ISMEMBER contains .TRUE. Otherwise, it is set to .FALSE.

# PXFSTAT

*Gets the status of a file*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSTAT (PATH, ILEN, JSTAT, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, JSTAT, IERROR
  END SUBROUTINE PXFSTAT
END INTERFACE
```

| | |
|---|---|
| PATH | file name |
| ILEN | length of PATH string |
| JSTAT | pointer to PXFSTAT structure |
| IERROR | return value |

### Description

This subroutine gets the status of a file.

### Output

If successful, IERROR is set to zero.

# PXFSTRGET

*Sets an integer component of a structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSTRGET (JHANDLE, COMPNAME, INDEX,
VALUE, ILEN, IERROR)
  INTEGER(4) JHANDLE, INDEX, ILEN, IERROR
  CHARACTER(LEN=*) COMPNAME, VALUE
  END SUBROUTINE PXFSTRGET
END INTERFACE
```

| | |
|---|---|
| JHANDLE | handle to the structure |
| COMPNAME | component name |
| INDEX | index of the component |
| VALUE | (out) where the value of the component is stored |
| ILEN | (out) length of VALUE string |
| IERROR | return value |

### Description

This subroutine sets an integer component of a structure.

### Output

If successful, IERROR is set to zero. ENONAME is set if no such component name exist in the structure.

# PXFSTRUCTCOPY

*Copies the contents of one structure to another*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSTRUCTCOPY (STRUCTNAME, JHANDLE1,
   JHANDLE2, IERROR)
  INTEGER(4) JHANDLE1, JHANDLE2, IERROR
  CHARACTER(LEN=*) STRUCTNAME
  END SUBROUTINE PXFSTRUCTCOPY
END INTERFACE
```

| | |
|---|---|
| STRUCTNAME | structure name |
| JHANDLE1 | handle to the structure to be copied??? |
| JHANDLE2 | handle to structure to be copied into??? |
| IERROR | return value |

### Description

This subroutine copies the contents of one structure to another.

### Output

If successful, IERROR is set to zero. ENONAME is an invalid structure name is specified.

# PXFSTRUCTCREATE

*Creates an instance of the specified structure*

### Prototype

```
INTERFACE
  SUBROUTINE PXFSTRUCTCREATE (STRUCTNAME, JHANDLE,
   IERROR)
  CHARACTER(LEN=*) STRUCTNAME
  INTEGER(4) JHANDLE, IERROR
  END SUBROUTINE PXFSTRUCTCREATE
END INTERFACE
```

| | |
|---|---|
| STRUCTNAME | structure name |
| JHANDLE | (out) handle to structure |
| IERROR | return value |

### Description

This subroutine creates an instance of the specified structure.

### Output

If successful, IERROR is set to zero.

# PXFSTRUCTFREE

*Deletes the instance of a structure*

### Prototype

```
INTERFACE
   SUBROUTINE PXFSTRUCTFREE (JHANDLE, IERROR)
   INTEGER(4) JHANDLE, IERROR
   END SUBROUTINE PXFSTRUCTFREE
END INTERFACE
```

JHANDLE          handle to structure

IERROR          return value

### Description

This subroutine deletes the instance of a structure.

### Output

If successful, IERROR is set to zero.

# PXFUCOMPARE

*Compares two unsigned numbers and returns the absolute value of their difference*

### Prototype

```
INTERFACE
   SUBROUTINE PXFUCOMPARE (I1, I2, ICMPR, IDIFF)
   INTEGER(4) I1, I2, ICMPR, IDIFF
   END SUBROUTINE PXFUCOMPARE
END INTERFACE
```

| | |
|---|---|
| I1, I2 | the two unsigned integers to compare |
| ICMPR | (output) result of comparison (-1,0,1) |
| IDIFF | (output) absolute value of the difference |

### Description

This subroutine compares two unsigned numbers and returns the absolute value of their difference.

### Output

The value of ICMPR upon completion:

| | |
|---|---|
| -1 | if I1 < I2 |
| 0 | if I1 = I2 |
| 1 | if I1 > I2 |

## PXFUMASK

*Sets and gets the file creation mask*

### Prototype

```
INTERFACE
  SUBROUTINE PXFUMASK (ICMASK, IPREVCMASK, IERROR)
  INTEGER(4) ICMASK, IPREVCMASK, IERROR
  END SUBROUTINE PXFUMASK
END INTERFACE
```

| | |
|---|---|
| ICMASK | set mask |
| IPREVCMASK | (out) previous mask |
| IERROR | return value |

### Description

This subroutine sets and gets the file creation mask.

**Output**

If successful, IERROR is set to zero.

# PXFUNLINK

*Removes the specified directory entry*

### Prototype

```
INTERFACE
  SUBROUTINE PXFUNLINK (PATH, ILEN, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, IERROR
  END SUBROUTINE PXFUNLINK
END INTERFACE
```

| | |
|---|---|
| PATH | name of the directory entry |
| ILEN | length of PATH string |
| IERROR | return value |

### Description

This subroutine removes the specified directory entry.

### Output

If successful, IERROR is set to zero.

## PXFUTIME

*Sets file access and modification times*

### Prototype

```
INTERFACE
  SUBROUTINE PXFUTIME (PATH, ILEN, JUTIMBUG, IERROR)
  CHARACTER(LEN=*) PATH
  INTEGER(4) ILEN, JUTIMBUF, IERROR
  END SUBROUTINE PXFUTIME
END INTERFACE
```

| | |
|---|---|
| PATH | file name |
| ILEN | length of PATH string |
| JUTIMBUG | pointer to PXFUTIMBUF structure |
| IERROR | return value |

### Description

This subroutine sets the file access and modification times.

### Output

If successful, IERROR is set to zero.

# PXFWAIT

*Waits for any child process*

### Prototype

```
INTERFACE
  SUBROUTINE PXFWAIT (ISTATE, IRETPID, IERROR)
  INTEGER(4) ISTAT, IRETPID, IERROR
  END SUBROUTINE PXFWAIT
END INTERFACE
```

| | |
|---|---|
| ISTATE | (output) status of child process |
| IRETPID | (input/output) the process ID to wait on when the function is called; the process ID of the stopped child process returned |
| IERROR | (output) error value |

### Description

This subroutine waits for any child process.

### Output

If successful, IERROR is set to zero.

# PXFWAITPID

*Waits for certain PIDs*

### Prototype

```
INTERFACE
  SUBROUTINE PXFWAITPID (IPID, ISTAT, IOPTIONS,
                         IRETPID, IERROR)
  INTEGER(4) IPID, ISTAT, IOPTIONS, IRETPID, IERROR
  END SUBROUTINE PXFWAITPID
```

```
END INTERFACE
```

| | |
|---|---|
| IPID | (input) specifies the PIDs to wait for |
| ISTAT | (output) status of child process |
| IOPTIONS | (input) on Unix, check `/usr/include/sys/wait.h` for possible values of the options. `WCONTINUED`, `WNOHANG`, `WNOWAIT`, `WUNTRACED`. On Win32 systems, check your MSVC installation, in the `include\process.h` file |
| IRETPID | (output) pid of the stopped child |
| IERROR | (output) error value |

### Description

This subroutine waits for certain PIDs.

### Output

The PID and status of the child process that triggered the `WAITPID` routine to stop waiting.

# PXFWIFEXITED

*.Determines if a child process has
exited.*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFWIFEXITED (ISTAT)
  INTEGER(4) ISTAT
  END FUNCTION PXFWIFEXITED
END INTERFACE
```

| | |
|---|---|
| ISTAT | an integer status obtained from `PXFWEXITSTATUS`. |

### Description

Tests whether a child process has exited.

### Output

Returns `.TRUE.` if the child process exited, `.FALSE.` otherwise.

# PXFWEXITSTATUS

*Returns the current status of the child process*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFWEXITSTATUS (ISTAT)
  INTEGER(4) ISTAT
  END FUNCTION PXFWEXITSTATUS
END INTERFACE
```

ISTAT          an `INTEGER*4` variable.

### Description

Returns the exit status of a child process created via `PXFFORK`. On input, `ISTAT` contains the process ID of the child process.

### Output

`ISTAT` is set to the exit status value of the child process which can then be interpreted by `PXFWIFEXITED`, `PXFWIFSIGNALED`, `PXFWTERMSIG`, or `PXFWIFSTOPPED`. If the child process is not found, the function returns `.FALSE.`.

# PXFWIFSIGNALED

*Determines if a child process has exited due to a signal*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFWIFSIGNALED (ISTAT)
  INTEGER(4) ISTAT
  END FUNCTION PXFWIFSIGNALED
END INTERFACE
```

ISTAT            the status variable returned from a previous call to
                 PXFWAIT or PXFWAITPID for a child process.

### Description

PXFWIFSIGNALED evaluates the status code returned from a previous call
to PXFWAITPID for a child process. If the child terminated due to an
missed signal, PXFWIFSIGNALED returns .TRUE..

### Output

IERROR is set to zero if a successful inquiry is made. The function result is
.TRUE. if child PID has exited, .FALSE. if the child is still active or
suspended.

# PXFWIFSTOPPED

*Determines if a child process is currently stopped or suspended*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFWIFSTOPPED (ISTAT)
  INTEGER(4) ISTAT
  END FUNCTION PXFWIFSTOPPED
END INTERFACE
```

ISTAT            the status variable returned from a previous call to
                 PXFWAIT or PXFWAITPID for a child process

### Description

PXFIFSTOPPED evaluates the input status code to determine whether a
particular child process is currently stopped.

### Output

IERRO is set to zero if a successful inquiry is made. The function result is
true if child PID has stopped, false otherwise.

# PXFWRITE

*Writes to a file*

### Prototype

```
INTERFACE
  SUBROUTINE PXFWRITE (IFILDES, BUF, NBYTE, NWRITTEN,
                              IERROR)
```

```
              INTEGER(4) IFILDES, NBYTE, NWRITTE, IERROR
              CHARACTER BUF(LEN=*)
              END SUBROUTINE PXFWRITE
            END INTERFACE
```

| | |
|---|---|
| IFILDES | descriptor to be written to |
| BUF | buffer to be written from |
| NBYTE | number of bytes to write |
| NWRITTEN | (output) number of bytes written |
| IERROR | (output) returned error code |

### Description

This subroutine writes output to a file.

### Output

If successful, IERROR is set to zero.

## PXFWSTOPSIG

*Gets the number of the signal that
caused the child process to stop*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFWSTOPSIG (ISTAT)
  INTEGER(4) ISTAT
  END FUNCTION PXFWSTOPSIG
END INTERFACE
```

| | |
|---|---|
| ISTAT | an integer value representing the child process ID on input, and the value of the signal that terminated the child process on output. |

### Description

This function gets the number of the signal that caused the child process to stop.

### Output

If successful, ISTAT contains the signal value and the function returns .TRUE.. Otherwise, the function returns .FALSE., and ISTAT is not changed.

# PXFWTERMSIG

*Gets the number of the signal that*
*caused termination of a child process*

### Prototype

```
INTERFACE
  LOGICAL(4) FUNCTION PXFWTERMSIG (ISTAT)
  INTEGER(4) ISTAT
  END FUNCTION PXFWTERMSIG
END INTERFACE
```

ISTAT                process ID on input, signal number on output

### Description

This function gets the number of the signal that caused termination of a child process defined by the process ID in ISTAT on input.

### Output

If successful, returns .TRUE. and the signal value in ISTAT. Otherwise returns .FALSE. and ISTAT is not changed.

# *QuickWin Library*   4

This chapter describes procedures that comprise the QuickWin library
(`libqwin.lib`). These procedures are made available to the compiler when
you invoke the `/MW` option and use the flib.fd include file.

The prototypes are described using the `INTERFACE` call, which provides the
required information to complete a call to the specified procedure. For
descriptions of the `INTERFACE` block and the `/MW` option, see the *Intel®
Fortran Compiler User's Guide* .

You can use the QuickWin library to compile programs into simple applications
for Windows which allow you to call graphic routines, load and save bitmaps,
perform edit and other operations available under windows.

To use the QuickWin library, your program must explicitly access this library
procedures with this statement:

```
INCLUDE
"<installation directory>\include\flib.fd"
```

If a procedure does not have a PROGRAM statement, then the above
statement must appear in each subprogram that makes graphics calls.

This chapter introduces the major groups of QuickWin library procedures
described in the following sections:

- QuickWin subroutines and functions
- Graphics subroutines and functions

# QuickWin Subroutines and Functions

This category includes subroutines and functions that provide windows control and configuration, enhance QuickWin applications and perform color conversion. To use the procedures of this group, add the following statement to the program unit containing the procedure:

```
INCLUDE
"<installation directory>\include\flib.fd"
```

**Table 4-1      QuickWin Routines**

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **Windows Control** | | |
| **FOCUSQQ**<br>result = FOCUSQQ (*iunit*) | Function | Sets focus to specified window. |
| **GETACTIVEQQ**<br>result = GETACTIVEQQ ( ) | Function | Returns the unit number of the currently active child. |
| **GETHWNDQQ**<br>result = GETHWNDQQ (*unit*) | Function | Converts the unit number into a Windows handle for functions that require it. |
| **GETWINDOWCONFIG**<br>result = GETWINDOWCONFIG (*wc*) | Function | Returns current window properties. |
| **GETWSIZEQQ**<br>result = GETWSIZEQQ (*unit*, *ireq*, *winfo*) | Function | Returns the size and position of a window. |
| **INQFOCUSQQ**<br>result = INQFOCUSQQ (*unit*) | Function | Determines which window has focus. |
| **SETACTIVEQQ**<br>result = SETACTIVEQQ (*unit*) | Function | Makes a child window active, but does not give it focus. |
| **SETWINDOWCONFIG**<br>result = SETWINDOWCONFIG (*wc*) | Function | Sets current window properties. |
| **SETWSIZEQQ**<br>result = SETWSIZEQQ (*unit*, *winfo*) | Function | Sets the size and position of a window. |

continued

**Table 4-1    QuickWin Routines** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **QuickWin Applications Enhancements** | | |
| **ABOUTBOXQQ**<br>result = ABOUTBOXQQ (*cstring*) | Function | Adds an About Box with customized text. |
| **APPENDMENUQQ**<br>result = APPENDMENUQQ (*menuID*, *flags*, *text*, *routine*) | Function | Appends a menu item. |
| **CLICKMENUQQ**<br>result = CLICKMENUQQ (item) | Function | Simulates the effect of clicking or selecting a menu item. |
| **DELETEMENUQQ**<br>result = DELETEMENUQQ (*menuID*, *itemID*) | Function | Deletes a menu item. |
| **GETEXITQQ**<br>result = GETEXITQQ ( ) | Function | Returns the setting for a QuickWin application's exit behavior. |
| **INCHARQQ**<br>result = INCHARQQ ( ) | Function | Reads a single character input from the keyboard and returns the ASCII value of that character without any buffering. |
| **INITIALSETTINGS**<br>result = INITIALSETTINGS ( ) | Function | Controls initial menu settings and initial frame window. |
| **INSERTMENUQQ**<br>result = INSERTMENUQQ (*menuID*, *itemID, flag, text, routine*) | Function | Inserts a menu item. |
| **MESSAGEBOXQQ**<br>result = MESSAGEBOXQQ (*msg, caption, mtype*) | Function | Displays a message box. |
| **MODIFYMENUFLAGSQQ**<br>result = MODIFYMENUFLAGSQQ (*menuID, itemID, flag*) | Function | Modifies a menu item's state. |

continued

**Table 4-1    QuickWin Routines** (continued)

| Name/Syntax | Subroutine / Function | Description |
| --- | --- | --- |
| **MODIFYMENUROUTINEQQ** <br> result = MODIFYMENUROUTINEQQ (*menuIdD, itemID, routine*) | Function | Modifies a menu item's callback routine. |
| **MODIFYMENUSTRINGQQ** <br> result = MODIFYMENUSTRINGQQ (*menuID, itemID, text*) | Function | Modifies a menu item's text string. |
| **REGISTERMOUSEEVENT** <br> result = REGISTERMOUSEEVENT (*unit, mouseevents, callbackroutine*) | Function | Registers the application defined routines to be called on mouse events. |
| **SETEXITQQ** <br> result = SETEXITQQ (*exitmode*) | Function | Sets a QuickWin application's exit behavior. |
| **SETMESSAGEQQ** <br> CALL SETMESSAGEQQ (*msg, id*) | Subroutine | Changes any QuickWin message, including status bar messages, state messages, and dialog box messages. |
| **SETWINDOWMENUQQ** <br> result = SETWINDOWMENUQQ (*menuID*) | Function | Sets the menu to which a list of current child window names are appended. |
| **UNREGISTERMOUSEEVENT** <br> result = UNREGISTERMOUSEEVENT (*unit, mouseevents*) | Function | Removes the routine registered by REGISTERMOUSEEVENT. |
| **WAITONMOUSEEVENT** <br> result = WAITONMOUSEEVENT (*mouseevents, keystate, x, y*) | Function | Blocks a return until a mouse event occurs. |
| **Color Conversion** | | |
| **INTEGERTORGB** <br> CALL INTEGERTORGB (*rgb, red, green, blue*) | Subroutine | Converts an RGB color value to its red, green, and blue components |

continue

**Table 4-1      QuickWin Routines** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **RGBTOINTEGER**<br>result = RGBTOINTEGER (*red, green, blue*) | Function | Converts integers specifying red, green, and blue color into an RGB integer (for use in RGB routines). |

# Graphics Procedures

The QuickWin run-time library helps you turn graphics programs into simple Windows applications. QuickWin applications support pixel-based graphics, real-coordinate graphics, text windows, character fonts, user-defined menus, mouse events, and editing (select/copy/paste) of text, graphics, or both.

To use the QuickWin library, invoke the compiler driver with the option /MW to link to the QuickWin library, libqwin.lib, and add the following statement to the program unit containing the QuickWin procedure(s):

INCLUDE
"<installation directory>\include\flib.fd"

When QuickWin is not being used, you link by default to the library libqwind.lib instead. This library consists of dummy versions of QuickWin functions to avoid unresolved references at link time.

Note that QuickWin applications cannot be DLLs and QuickWin cannot be linked with runtime routines that are in DLLs.

Table 4-2 summarizes the Graphics procedures.  When writing your applications, you can use any case.

**Table 4-2        Graphics Routines and Functions Summary**

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **ARC**, **ARC_W**<br>result = ARC (*x1, y1, x2, y2, x3, y3, x4, y4*)<br>result = ARC_W (*wx1, wy1, wx2, wy2, wx3, wy3, wx4, wy4*) | Function | Draws an arc. |
| **CLEARSCREEN**<br>CALL CLEARSCREEN (*area*) | Subroutine | Clears the screen, viewport, or text window. |
| **DISPLAYCURSOR**<br>result = DISPLAYCURSOR (*toggle*) | Function | Graphics Function  Turns the cursor off and on. |
| **ELLIPSE**, **ELLIPSE_W**<br>result = ELLIPSE (*control, x1, y1, x2, y2*)<br>result = ELLIPSE_W (*control, wx1, wy1, wx2, wy2*) | Function | Draws an ellipse or circle |
| **FLOODFILL**, **FLOODFILL_W**<br>result = FLOODFILL (*x, y, bcolor*)<br>result = FLOODFILL_W (*wx, wy, bcolor*) | Function | Fills an enclosed area of the screen with the current color index, using the current fill mask. |
| **FLOODFILLRGB**, **FLOODFILLRGB_W**<br>result = FLOODFILLRGB (*x, y, color*)<br>result = FLOODFILLRGB_W (*wx, wy, color*) | Function | Fills an enclosed area of the screen with the current RGB color, using the current. |
| **GETARCINFO**<br>result = GETARCINFO (*pstart, pend, ppaint*) | Function | Determines the end points of the most recently drawn arc or pie. |
| **GETBKCOLOR**<br>result = GETBKCOLOR ( ) | Function | Returns the current background color index. |

**Table 4-2      Graphics Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
| --- | --- | --- |
| **GETBKCOLORRGB**<br>result = GETBKCOLORRGB ( ) | Function | Returns the current background RGB color. |
| **GETCOLOR**<br>result = GETCOLOR ( ) | Function | Returns the current color index. |
| **GETCOLORRGB**<br>result = GETCOLORRGB ( ) | Function | Returns the current RGB color. |
| **GETCURRENTPOSITION,**<br>**GETCURRENTPOSITION_W**<br><br>CALL GETCURRENTPOSITION (t)<br>CALL GETCURRENTPOSITION_W (wt) | Subroutine | Returns the coordinates of the current graphics-output position. |
| **GETFILLMASK**<br>CALL GETFILLMASK (*mask*) | Subroutine | Returns the current fill mask |
| **GETFONTINFO**<br>result = GETFONTINFO (*font*) | Function | Returns the current font characteristics. |
| **GETGTEXTEXTENT**<br>result = GETGTEXTEXTENT (*text*) | Function | Determines the width of the specified text in the current font. |
| **GETGTEXTROTATION**<br>result = GETGTEXTROTATION ( ) | Function | Get the current text rotation angle. |
| **GETIMAGE, GETIMAGE_W**<br><br>CALL GETIMAGE (*x1, y1, x2, y2, image*)<br>CALL GETIMAGE_W (*wx1, wy1, wx2, wy2, image*) | Subroutine | Stores a screen image in memory. |
| **GETLINESTYLE**<br>result = GETLINESTYLE ( ) | Function | Returns the current line style. |
| **GETPHYSCOORD**<br>CALL GETPHYSCOORD (*x, y, t*) | Subroutine | Converts viewport coordinates to physical coordinates. |

continued

**Table 4-2      Graphics Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **GETPIXEL, GETPIXEL_W**<br><br>result = GETPIXEL (*x, y*)<br>result = GETPIXEL_W (*wx, wy*) | Function | Returns a pixel's color index. |
| **GETPIXELRGB, GETPIXELRGB_W**<br><br>result = GETPIXELRGB (*x, y*)<br>result = GETPIXELRGB_W (*wx, wy*) | Function | Returns a pixel's RGB color. |
| **GETPIXELS**<br>CALL GETPIXELS (*n, x, y, color*) | Function | Returns the color indices of multiple pixels. |
| **GETPIXELSRGB**<br>CALL GETPIXELSRGB (*n, x, y, color*) | Function | Returns the RGB colors of multiple pixels. |
| **GETTEXTCOLOR**<br>result = GETTEXTCOLOR ( ) | Function | Returns the current text color index |
| **GETTEXTCOLORRGB**<br>result = GETTEXTCOLORRGB ( ) | Function | Returns the current text RGB color. |
| **GETTEXTPOSITION**<br>CALL GETTEXTPOSITION (t) | Subroutine | Returns the current text-output position. |
| **GETTEXTWINDOW**<br>CALL GETTEXTWINDOW (*r1, c1, r2, c2*) | Subroutine | Returns the boundaries of the current text window. |
| **GETVIEWCOORD, GETVIEWCOORD_W**<br><br>CALL GETVIEWCOORD (*x, y, t*)<br>CALL GETVIEWCOORD_W (*wx, wy, wt*) | Subroutine | Converts physical or window coordinates to viewport coordinates. |
| **GETWINDOWCOORD**<br>CALL GETWINDOWCOORD (*x, y, wt*) | Subroutine | Converts viewport coordinates to window coordinates. |
| **GETWRITEMODE**<br>result = GETWRITEMODE ( ) | Function | Returns the logical write mode for lines. |

**Table 4-2    Graphics Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **GRSTATUS**<br>result = GRSTATUS ( ) | Function | Returns the status (success or failure) of the most recently called graphics routine. |
| **IMAGESIZE, IMAGESIZE_W**<br>result = IMAGESIZE (*x1, y1, x2, y2*)<br>result = IMAGESIZE_W (*wx1, wy1, wx2, wy2*) | Function | Returns image size in bytes. |
| **INITIALIZEFONTS**<br>result = INITIALIZEFONTS ( ) | Function | Initializes the font library. |
| **LINETO, LINETO_W**<br>result = LINETO (*x, y*)<br>result = LINETO_W (*wx, wy*) | Function | Draws a line from the current position to a specified point. |
| **LOADIMAGE, LOADIMAGE_W**<br>result = LOADIMAGE (*filename, xcoord, ycoord*)<br>result = LOADIMAGE_W (*filename, wxcoord, wycoord*) | Function | Reads a Windows bitmap file (.BMP) and displays it at the specified location. |
| **MOVETO, MOVETO_W**<br>CALL MOVETO (*x, y, t*)<br>CALL MOVETO_W (*wx, wy, w*) | Subroutine | Moves the current position to the specified point. |
| **OUTGTEXT**<br>CALL OUTGTEXT (*text*) | Subroutine | Sends text in the current font to the screen at the current position. |
| **OUTTEXT**<br>CALL OUTTEXT (*text*) | Subroutine | Sends text to the screen at the current position. |

**Table 4-2** **Graphics Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **PIE**, *PIE_W*<br><br>result = PIE (*i, x1, y1, x2, y2, x3, y3, x4, y4*) <br>result = PIE_W (*i, wx1, wy1, wx2, wy2, wx3, wy3, wx4, wy4*) | Function | Draws a pie slice. |
| **POLYGON**, **POLYGON_W**<br><br>result = POLYGON (*control, ppoints, cpoints*)<br>result = POLYGON_W (*control, wppoints, cpoints*) | Function | Draws a polygon. |
| **PUTIMAGE**, **PUTIMAGE_W**<br><br>CALL PUTIMAGE (*x, y, image, action*)<br>CALL PUTIMAGE_W (*wx, wy, image, action*) | Subroutine | Retrieves an image from memory and displays it. |
| **RECTANGLE**, **RECTANGLE_W**<br><br>result = RECTANGLE (*control, x1, y1, x2, y2*)<br>result = RECTANGLE_W (control, wx1, wy1, wx2, wy2) | Function | Draws a rectangle. |
| **REMAPALLPALETTERGB**<br><br>result = REMAPALLPALETTERGB (*colors*) | Function | Remaps a set of RGB color values to indices recognized by the current video configuration. |
| **REMAPPALETTERGB**<br><br>result = REMAPPALETTERGB (*index, color*) | Function | Remaps a single RGB color value to a color index. |

**Table 4-2**     **Graphics Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **SAVEIMAGE, SAVEIMAGE_W**<br><br>result = SAVEIMAGE (*filename, ulxcoord, ulycoord, lrxcoord, lrycoord*)<br>result = SAVEIMAGE_W (*filename, ulwxcoord, ulwycoord, lrwxcoord, lrwycoord*) | Function | Captures a screen image and saves it as a Windows bitmap file. |
| **SCROLLTEXTWINDOW**<br><br>CALL SCROLLTEXTWINDOW (*rows*) | Function | Scrolls the contents of a text window. |
| **SETBKCOLOR**<br><br>result = SETBKCOLOR (*color*) | Function | Sets the current background color. |
| **SETBKCOLORRGB**<br><br>result = SETBKCOLORRGB (*color*) | Function | Sets the current background color to a direct color value rather than an index to a defined palette. |
| **SETCLIPRGN**<br><br>CALL SETCLIPRGN (*x1, y1, x2, x2*) | Subroutine | Limits graphics output to a part of the screen. |
| **SETCOLOR**<br><br>result = SETCOLOR (*color*) | Function | Sets the current color to a new color index. |
| **SETCOLORRGB**<br><br>result = SETCOLORRGB (*color*) | Function | Sets the current color to a direct color value rather than an index to a defined palette. |
| **SETFILLMASK**<br><br>CALL SETFILLMASK (*mask*) | Subroutine | Changes the current fill mask to a new pattern. |
| **SETFONT**<br><br>result = SETFONT (*options*) | Function | Finds a single font matching the specified characteristics and assigns it to OUTGTEXT. |

**Table 4-2    Graphics Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **SETGTEXTROTATION**<br>CALL SETGTEXTROTATION (*degrees*) | Subroutine | Sets the direction in which text is written to the specified angle. |
| **SETLINESTYLE**<br>CALL SETLINESTYLE (mask) | Subroutine | Changes the current line style. |
| **SETPIXEL, SETPIXEL_W**<br>result = SETPIXEL (*x, y*)<br>result = SETPIXEL_W (*wx, wy*) | Function | Sets color of a pixel at a specified location. |
| **SETPIXELRGB, SETPIXELRGB_W**<br>result = SETPIXELRGB (*x, y, color*)<br>result = SETPIXELRGB_W (*wx, wy, color*) | Function | Sets RGB color of a pixel at a specified location. |
| **SETPIXELS**<br>CALL SETPIXELS (*n, x, y, color*) | Subroutine | Sets the color indices of multiple pixels. |
| **SETPIXELSRGB**<br>CALL SETPIXELSRGB (*n, x, y, color*) | Subroutine | Sets the RGB color of multiple pixels. |
| **SETTEXTCOLOR**<br>result = SETTEXTCOLOR (*index*) | Function | Sets the current text color to a new color index. |
| **SETTEXTCOLORRGB**<br>result = SETTEXTCOLORRGB (*color*) | Function | Sets the current text color to a direct color value rather than an index to a defined palette. |
| **SETTEXTPOSITION**<br>CALL SETTEXTPOSITION (*row, column, t*) | Subroutine | Changes the current text position. |
| **SETTEXTWINDOW**<br>CALL SETTEXTWINDOW (*r1, c1, r2, c2*) | Subroutine | Sets the current text display window. |

**Table 4-2    Graphics Routines and Functions Summary** (continued)

| Name/Syntax | Subroutine / Function | Description |
|---|---|---|
| **SETVIEWORG**<br>CALL SETVIEWORG (*x, y, t*) | Subroutine | Positions the viewport coordinate origin. |
| **SETVIEWPORT**<br>CALL SETVIEWPORT (*x1, y1, x2, y2*) | Subroutine | Defines the size and screen position of the viewport. |
| **SETWINDOW**<br>result = SETWINDOW (*finvert, wx1, wy1, wx2, wy2*) | Function | Defines the window coordinate system. |
| **SETWRITEMODE**<br>result = SETWRITEMODE (*wmode*) | Function | Changes the current logical write mode for lines. |
| **WRAPON**<br>result = WRAPON (*option*) | Function | Turns line wrapping on or off. |

# Graphic Function Descriptions

# ARC

*Draws elliptical arcs using the current graphics color*

### Prototype

```
INTERFACE
  FUNCTION ARC(X1,Y1,X2,Y2,X3,Y3,X4,Y4)
    INTEGER(2) ARC,X1,Y1,X2,Y2,X3,Y3,X4,Y4
  END FUNCTION
END INTERFACE
```

| X1, Y1 | Input. INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle. |
| X2, Y2 | Input. INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle. |
| X3, Y3 | Input. INTEGER(2). Viewport coordinates of start vector. |
| X4, Y4 | Input. INTEGER(2). Viewport coordinates of end vector. |

### Description

This function draws elliptical arcs using the current graphics color. The center of the arc is the center of the bounding rectangle defined by the points (X1, Y1) and (X2, Y2).

The arc starts where it intersects an imaginary line extending from the center of the arc through (X3, Y3). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (X4, Y4).

ARC uses the view-coordinate system. The arc is drawn using the current color.

### Output

The result type is INTEGER(2). It is nonzero if successful; otherwise, 0. If the arc is clipped or partially out of bounds, the arc is considered successfully drawn and the return is 1. If the arc is drawn completely out of bounds, the return is 0.

# ARC_W

*Draws elliptical arcs using the current graphics color*

## Prototype

```
INTERFACE
  FUNCTION ARC_W(WX1,WY1,WX2,WY2,WX3,WY3,WX4,WY4)
  INTEGER(2) ARC_W
  DOUBLE PRECISION WX1,WY1,WX2,WY2,WX3,WY3,WX4,WY4
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| WX1,WY1 | Input. REAL(8). Window coordinates for upper-left corner of bounding rectangle. |
| WX2,WY2 | Input. REAL(8). Window coordinates for lower-right corner of bounding rectangle. |
| WX3,WY3 | Input. REAL(8). Window coordinates of start vector. |
| WX4,WY4 | Input. REAL(8). Window coordinates of end vector. |

## Description

This function draws elliptical arcs using the current graphics color. The center of the arc is the center of the bounding rectangle defined by the points (WX1, WY1) and (WX2, WY2).

The arc starts where it intersects an imaginary line extending from the center of the arc through (WX3, WY3). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (WX4, WY4).

ARC_W uses the view-coordinate system. The arc is drawn using the current color.

### Output

The result type is INTEGER(2). It is nonzero if successful; otherwise, 0. If the arc is clipped or partially out of bounds, the arc is considered successfully drawn and the return is 1. If the arc is drawn completely out of bounds, the return is 0.

# GETARCINFO

*Determines the endpoints (in viewport coordinates) of the most recently drawn arc or pie.*

### Prototype

```
INTERFACE
 FUNCTION GETARCINGO(LPSTART,LPEND,LPPAINT)
 INTEGER(2) GETARCHINFO
 STRUCTURE /XYCOORD/
  INTEGER(2) XCOORD
  INTEGER(2) YCOORD
 END STRUCTURE
 RECORD /XYCOORD/ LPSTART
 RECORD /XYCOORD/ LPEND
 RECORD /XYCOORD/ LPPAINT
 END FUNCTION
END INTERFACE
```

| | |
|---|---|
| LPSTART | Output. Derived type XYCOORD. Viewport coordinates of the starting point of the arc. |
| LPEND | Output. Derived type XYCOORD. Viewport coordinates of the end point of the arc. |
| LPPAINT | Output. Derived type XYCOORD. Viewport coordinates of the point at which the fill begins. |

### Description

`GETARCINFO` updates the `LPSTART` and `LPEND XYCOORD` derived types to contain the endpoints (in viewport coordinates) of the arc drawn by the most recent call to the `ARC` or `PIE` functions. The `XYCOORD` derived type, defined is:

```
TYPE XYCOORD
  INTEGER(2) XCOORD
  INTEGER(2) YCOORD
END TYPE XYCOORD
```

The returned value in `LPPAINT` specifies a point from which a pie can be filled. You can use this to fill a pie in a color different from the border color. After a call to `GETARCINFO`, change colors using `SETCOLORRGB`. Use the new color, along with the coordinates in `LPPAINT`, as arguments for the `FLOODFILLRGB` function.

### Output

The result type is `INTEGER(2)`. The result is nonzero if successful. The result is zero if neither the `ARC` nor the `PIE` function has been successfully called since the last time `CLEARSCREEN` or `SETWINDOWCONFIG` was successfully called, or since a new viewport was selected.

## CLEARSCREEN

*Erases the target area and fills it with*
*the current background color*

### Prototype

```
INTERFACE
  SUBROUTINE CLEARSCREEN(AREA)
  INTEGER(2) AREA
  END SUBROUTINE
```

```
END INTERFACE
```

AREA                Input. INTEGER(4). Identifies the target area.

### Description

Parameter AREA identifies the target area. Must be one of the following symbolic constants (defined in flib.fd):

$GCLEARSCREEN

Clears the entire screen.

$GVIEWPORT      Clears only the current viewport.

$GWINDOW        Clears only the current text window (set with SETTEXTWINDOW).

All pixels in the target area are set to the color specified with SETBKCOLORRGB. The default color is black.

### Output

None

# DISPLAYCURSOR

*Controls cursor visibility*

### Prototype

```
INTERFACE
    FUNCTION DISPLAYCURSOR(TOGGLE)
    INTEGER(2) DISPLAYCURSOR, TOGGLE
    END FUNCTION
END INTERFACE
```

TOGGLE              Input. INTEGER(2). Constant that defines the cursor state. Has two values:

$GCURSOROFF

> Makes the cursor invisible regardless of its current shape and mode.

$GCURSORON

> Makes the cursor always visible in graphics mode.

### Description

The function controls cursor visibility. Has these two values:

Cursor settings hold only for the currently active child window. You need to call `DISPLAYCURSOR` for each window in which you want the cursor to be visible.

A call to `SETWINDOWCONFIG` turns off the cursor.

### Output

The result is the previous value of `TOGGLE`.

# ELLIPSE

*Draws a circle or an ellipse using the current graphics color.*

### Prototype

```
INTERFACE
    FUNCTION ELLIPSE(CONTROL, X1, Y1, X2, Y2)
    INTEGER(2) ELLIPSE, CONTROL, X1, Y1, X2, Y2
    END FUNCTION
END INTERFACE
```

CONTROL          Input. `INTEGER(2)`. Fill flag. Can be one of the following symbolic constants:

$GFILLINTERIOR

Fills the figure using the current color and fill mask.

$GBORDER    Does not fill the figure.

X1, Y1    Input. INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.

X2, Y2    Input. INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.

## Description

When you use ELLIPSE, the center of the ellipse is the center of the bounding rectangle defined by the viewport-coordinate points (X1, Y1) and (X2, Y2). If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

The control option given by $GFILLINTERIOR is equivalent to a subsequent call to the FLOODFILLRGB function using the center of the ellipse as the start point and the current color (set by SETCOLORRGB) as the boundary color. The border is drawn in the current color and line style.

## Output

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the ellipse is clipped or partially out of bounds, the ellipse is considered successfully drawn, and the return is 1. If the ellipse is drawn completely out of bounds, the return is 0.

# ELLIPS_W

*Draws a circle or an ellipse using the current graphics color.*

### Prototype

```
INTERFACE
   FUNCTION ELLIPSE_W(CONTROL, WX1, WY1, WX2, WY2)
   INTEGER(2) ELLIPSE_W, CONTROL
   DOUBLE PRECISION WX1, WY1, WX2, WY2
   END FUNCTION
END INTERFACE
```

CONTROL Input. INTEGER(2). Fill flag. Can be one of the following symbolic constants:

$GFILLINTERIOR
Fills the figure using the current color and fill mask.

$GBORDER Does not fill the figure.

WX1, WY1 Input. REAL(8). Window coordinates for upper-left corner of bounding rectangle.

WX2, WY2 Input. REAL(8). Window coordinates for lower-right corner of bounding rectangle.

### Description

When you use ELLIPSE_W, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points (WX1, WY1) and (WX2, WY2). If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

The control option given by $GFILLINTERIOR is equivalent to a subsequent call to the FLOODFILLRGB function using the center of the ellipse as the start point and the current color (set by SETCOLORRGB) as the boundary color.

The border is drawn in the current color and line style.

### Output

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the ellipse is clipped or partially out of bounds, the ellipse is considered successfully drawn, and the return is 1. If the ellipse is drawn completely out of bounds, the return is 0.

# FlOODFILL

*Fills an area using the current color index and fill mask.*

### Prototype

```
INTERFACE
   FUNCTION FLOODFILL(X,Y,BOUNDARY)
      INTEGER(2) FLOODFILL, X, Y, BOUNDARY
   END FUNCTION
END INTERFACE
```

| | |
|---|---|
| X, Y | Input. INTEGER(2). Viewport coordinates for fill starting point. |
| BCOLOR | Input. INTEGER(2). Color index of the boundary color. |

### Description

FLOODFILL begins filling at the viewport-coordinate point (X, Y). The fill color used by FLOODFILL is set by SETCOLOR. You can obtain the current fill color index by calling GETCOLOR. These functions allow access only to the colors in the palette (256 or less). To access all available colors on a VGA (262,144 colors) or a true color system, use the RGB functions FLOODFILLRGB and FLOODFILLRGB_W.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current graphics color index set by SETCOLOR. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color BCOLOR.

## Output

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color BCOLOR, or if the starting point lies outside the clipping region).

# FLOODFILL_W

*Fills an area using the current color
index and fill mask.*

## Prototype

```
INTERFACE
 FUNCTION FLOODFILL_W(WX1,WY1,BOUNDARY)
 INTEGER(2) FLOODFILL_W, BOUNDARY
 DOUBLE PRECISION WX1, WY1
 END FUNCTION
END INTERFACE
```

WX1, WY1        Input. REAL(8). Window coordinates for fill starting point.

BCOLOR          Input. INTEGER(2). Color index of the boundary color.

## Description

FLOODFILL_W begins filling at the window-coordinate point (WX1, WY1). The fill color used by FLOODFILL_W is set by SETCOLOR. You can obtain the current fill color index by calling GETCOLOR. These functions allow

access only to the colors in the palette (256 or less). To access all available colors on a VGA (262,144 colors) or a true color system, use the RGB functions FLOODFILLRGB and FLOODFILLRGB_W.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current graphics color index set by SETCOLOR. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color BCOLOR.

### Output

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color BCOLOR, or if the starting point lies outside the clipping region).

# FLOODFILLRGB

*Fills an area using the current Red-Green-Blue (RGB) color and fill mask.*

### Prototype

```
INTERFACE
  FUNCTION FLOODFILLRGB(X,Y,BCOLOR)
    INTEGER(2) FLOODFILLRGB,X,Y
    INTEGER(4) BCOLOR
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| X, Y | Input. INTEGER(2). Viewport coordinates for fill starting point. |
| BCOLOR | Input. INTEGER(4). RGB value of the boundary color. |

### Description

FLOODFILLRGB begins filling at the viewport-coordinate point (X, Y). The fill color used by FLOODFILLRGB is set by SETCOLORRGB. You can obtain the current fill color by calling GETCOLORRGB.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current color set by SETCOLORRGB. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color *color*.

### Output

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color BCOLOR, or if the starting point lies outside the clipping region).

# FLOODFILLRGB_W

*Fills an area using the current*
*Red-Green-Blue (RGB) color and fill mask.*

### Prototype

```
INTERFACE
   FUNCTION FLOODFILLRGB_W(WX,WY,BCOLOR)
      INTEGER(2) FLOODFILLRGB_W
      DOUBLE PRECISIPON WX,WY
      INTEGER(4) BCOLOR
   END FUNCTION
END INTERFACE
```

WX, WY          Input. REAL(8). Window coordinates for fill starting point.

BCOLOR          Input. INTEGER(4). RGB value of the boundary color.

### Description

FLOODFILLRGB_W begins filling at the window-coordinate point (WX, WY). The fill color used by FLOODFILLRGB_W is set by SETCOLORRGB. You can obtain the current fill color by calling GETCOLORRGB.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current color set by SETCOLORRGB. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color, BCOLOR.

### Output

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color *color*, or if the starting point lies outside the clipping region).

# GETBKCOLOR

*Gets the current background color index*
*for both text and graphics output.*

### Prototype

```
INTERFACE
  FUNCTION GETBKCOLOR()
    INTEGER(4) GENBKCOLOR
  END FUNCTION
END INTERFACE
```

### Description

GETBKCOLOR returns the current background color index for both text and graphics, as set with SETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR and returned with GETTEXTCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETBKCOLORRGB, SETCOLORRGB, and SETTEXTCOLORRGB.

Generally, INTEGER(4) color arguments refer to color values and INTEGER(2) color arguments refer to color indexes. The two exceptions are GETBKCOLOR and SETBKCOLOR. The default background index is 0, which is associated with black unless the user remaps the palette with REMAPPALETTERGB.

### Output

The result type is INTEGER(4). The result is the current background color index.

# GETCOLOR

*Gets the current graphics color index.*

### Prototype

```
INTERFACE
  FUNCTION GETCOLOR()
    INTEGER(2) GETCOLOR
  END FUNCTION
END INTERFACE
```

### Description

GETCOLOR returns the current color index used for graphics over the background color as set with SETCOLOR. The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR and returned with GETTEXTCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB.

### Output

The result type is INTEGER(2). The result is the current color index, if successful; otherwise, - 1.

# GETCURRENTPOSITION

*Get the coordinates of the current graphics position.*

### Prototype

```
INTERFACE
  SUBROUTINE GETCURRENTPOSITION(S)
    STRUCTURE /XYCOORD/
        INTEGER(2) XCOORD, YCOORD
        END STRUCTURE
      RECORD /XYCOORD/ S
    END SUBROUTINE
  END INTERFACE
```

T                  Output. Derived type xycoord. Viewport coordinates of current graphics position. The derived type XYCOORD is defined as follows:

```
TYPE XYCOORD
```

```
      INTEGER(2) XCOORD    ! x-coordinate
      INTEGER(2) YCOORD    ! y-coordinate
END TYPE XYCOORD
```

### Description

LINETO, MOVETO, and OUTGTEXT all change the current graphics position. It is in the center of the screen when a window is created.

Graphics output starts at the current graphics position returned by GETCURRENTPOSITION. This position is not related to normal text output (from OUTTEXT or WRITE, for example), which begins at the current text position (see SETTEXTPOSITION). It does, however, affect graphics text output from OUTGTEXT.

# GETCURRENTPOSITION_W

*Get the coordinates of the current
graphics position.*

### Prototype

```
     INTERFACE
       SUBROUTINE GETCURRENTPOSITION_W(S)
         STRUCTURE /WXYCOORD/
             DOUBLE PRECISION WX, YX
             END STRUCTURE
         RECORD /WXYCOORD/ s
       END SUBROUTINE
     END INTERFACE
```

WT              Output. Derived type WXYCOORD. Window coordinates of current graphics position. The derived type wxycoord is defined as follows:

TYPE WXYCOORD

```
  REAL(8) WX    ! x-coordinate
  REAL(8) WY    ! y-coordinate
END TYPE WXYCOORD
```

### Description

LINETO, MOVETO, and OUTGTEXT all change the current graphics position. It is in the center of the screen when a window is created.

Graphics output starts at the current graphics position returned by GETCURRENTPOSITION_W. This position is not related to normal text output (from OUTTEXT or WRITE, for example), which begins at the current text position (see SETTEXTPOSITION). It does, however, affect graphics text output from OUTGTEXT.

# GETFILLMASK

*Returns the current pattern used to fill shapes.*

### Prototype

```
INTERFACE
  SUBROUTINE GETFILL(MASK)
    INTEGER(1) MASK(8)
  END SUBROUTINE
END INTERFACE
```

MASK              Output. INTEGER(1). One-dimensional array of length 8.

### Description

There are 8 bytes in MASK, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of MASK becomes the top 8 bits of the pattern, and the eighth element (byte) of MASK becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of 0 are unchanged. The current graphics color is set with SETCOLORRGB or SETCOLOR. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with SETFILLMASK), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (FLOODFILLRGB, PIE, ELLIPSE, POLYGON, and RECTANGLE).

# GETIMAGE

*Stores the screen image defined by a specified bounding rectangle.*

## Prototype

```
INTERFACE
  SUBROUTINE GETIMAGE(X1,Y1,X2,Y2,IMAGE)
    INTEGER(2) X1,Y1,X2,Y2
    INTEGER(1) IMAGE(*)
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| X1, Y1 | Input. INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle. |
| X2, Y2 | Input. INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle. |
| IMAGE | Output. INTEGER(1). Array of single-byte integers. Stored image buffer. |

## Description

GETIMAGE defines the bounding rectangle in viewport-coordinate points (X1, Y1) and (X2, Y2).

The buffer used to store the image must be large enough to hold it. You can determine the image size by calling `IMAGESIZE` at run time, or by using the formula described under `IMAGESIZE`. After you have determined the image size, you can dimension the buffer accordingly.

# GETIMAGE_W

*Stores the screen image defined by a specified bounding rectangle.*

### Prototype

```
INTERFACE
   SUBROUTINE GETIMAGE_W(WX1,WY1,WX2,WY2,IMAGE)
      double precision WX1,WY1,WX2,WY2
      INTEGER(1) IMAGE(*)
!MS$ ATTRIBUTES REFERENCE ::   IMAGE
      END SUBROUTINE
   END INTERFACE
```

| | |
|---|---|
| WX1, WY1 | Input. REAL(8). Window coordinates for upper-left corner of bounding rectangle. |
| WX2, WY2 | Input. REAL(8). Window coordinates for lower-right corner of bounding rectangle. |
| IMAGE | Output. INTEGER(1). Array of single-byte integers. Stored image buffer. |

### Description

`GETIMAGE_W` defines the bounding rectangle in window-coordinate points (WX1, WY1) and (WX2, WY2).

The buffer used to store the image must be large enough to hold it. You can determine the image size by calling `IMAGESIZE` at run time, or by using the formula described under `IMAGESIZE`. After you have determined the image size, you can dimension the buffer accordingly.

# GETLINESTYLE

*Returns the current graphics line style.*

### Prototype

```
INTERFACE
  FUNCTION GETLINESTYLE()
     INTEGER(2) GETLINESTYLE
  END FUNCTION
END INTERFACE
```

### Description

`GETLINESTYLE` retrieves the mask (line style) used for line drawing. The mask is a 16-bit number, where each bit represents a pixel in the line being drawn.

If a bit is 1, the corresponding pixel is colored according to the current graphics color and logical write mode; if a bit is 0, the corresponding pixel is left unchanged. The mask is repeated for the entire length of the line. The default mask is `#FFFF` (a solid line). A dashed line can be represented by `#FF00` (long dashes) or `#F0F0` (short dashes).

The line style is set with `SETLINESTYLE`. The current graphics color is set with `SETCOLORRGB` or `SETCOLOR`. `SETWRITEMODE` affects how the line is displayed.

The line style retrieved by `GETLINESTYLE` affects the drawing of straight lines as in `LINETO`, `POLYGON` and `RECTANGLE`, but not the drawing of curved lines as in `ARC`, `ELLIPSE` or `PIE`.

**Output**

The result type is INTEGER(2). The result is the current line style.

# GETPHYSCOORD

*Translates viewport coordinates to
physical coordinates.*

**Prototype**

```
INTERFACE
  SUBROUTINE GETPHYSCOORD(X,Y,S)
    INTEGER(2) X, Y
    STRUCTURE /XYCOORD/
        INTEGER(2) XCOORD, YCOORD
    END STRUCTURE
    RECORD /XYCOORD/ S
  END SUBROUTINE
END INTERFACE
```

X, Y          Input. INTEGER(2). Viewport coordinates to be
              translated to physical coordinates.

S             Output. Derived Type XYCOORD. Physical coordinates
              of the input viewport position.

```
TYPE xycoord
  INTEGER(2) XCOORD    ! x-coordinate
  INTEGER(2) YCOORD    ! y-coordinate
END TYPE XYCOORD
```

### Description

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with `SETVIEWPORT`. Both take integer coordinate values. Window coordinates refer to a window sized with `SETWINDOW` or `SETWSIZEQQ`. Window coordinates are floating-point values and allow easy scaling of data to the window area.

# GETPIXEL

*Returns the color index of the pixel at a specified location.*

### Prototype

```
INTERFACE
  FUNCTION GETPRIXEL(X,Y)
   INTEGER(2) GETPIXEL,X,Y
  END FUNCTION
END INTERFACE
```

X, Y      Input. `INTEGER(2)`. Viewport coordinates for pixel position.

### Description

Color routines without the `RGB` suffix, such as `GETPIXEL`, use color indexes, not true color values, and limit you to colors in the palette, at most 256. To access all system colors, use `SETPIXELRGB` to specify an explicit Red-Green-Blue value and retrieve the value with `GETPIXELRGB`.

### Output

The result type is `INTEGER(2)`. The result is the pixel color index if successful; otherwise, -1 (if the pixel lies outside the clipping region, for example).

# GETPIXEL_W

*Returns the color index of the pixel at a specified location.*

### Prototype

```
INTERFACE
  FUNCTION GETPIXEL_W(WX,WY)
     INTEGER(2) GETPIXEL_W
     DOUBLE PRECISION WX,WY
  END FUNCTION
END INTERFACE
```

WX, WY          Input. REAL(8). Window coordinates for pixel position.

### Description

Color routines without the RGB suffix, such as GETPIXEL_W, use color indexes, not true color values, and limit you to colors in the palette, at most 256. To access all system colors, use SETPIXELRGB to specify an explicit Red-Green-Blue value and retrieve the value with GETPIXELRGB.

### Output

The result type is INTEGER(2). The result is the pixel color index if successful; otherwise, -1 (if the pixel lies outside the clipping region, for example).

# GETPIXELS

*Gets the color indexes of multiple pixels.*

### Prototype

```
INTERFACE
  SUBROUTINE GETPIXELS(N, X, Y, C)
    INTEGER(4) N      ! input : size of arrays
    INTEGER(2) X(*)   ! input : x coordinates
    INTEGER(2) Y(*)   ! input : y coordinates
    INTEGER(2) C(*)   ! input : palette indices
  END SUBROUTINE
END INTERFACE
```

N            Input. `INTEGER(4)`. Number of pixels to get. Sets the number of elements in the other arguments.

X, Y         Input. `INTEGER(2)`. Parallel arrays containing viewport coordinates of pixels to get.

C            Output. `INTEGER(2)`. Array to be filled with the color indexes of the pixels at *x* and *y*.

### Description

GETPIXELS fills in the array COLOR with color indexes of the pixels specified by the two input arrays X and Y. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

If the pixel is outside the clipping region, the value placed in the *color* array is undefined. Calls to GETPIXELS with N less than 1 are ignored. GETPIXELS is a much faster way to acquire multiple pixel color indexes than individual calls to GETPIXEL.

The range of possible pixel color index values is determined by the current video mode and palette, at most 256 colors. To access all system colors you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function such as SETPIXELSRGB and retrieve the value with GETPIXELSRGB, rather than a palette index with a non-RGB color function.

# GETTEXTCOLOR

*Gets the current text color index.*

### Prototype

```
INTERFACE
   FUNCTION GETTEXTCOLOR()
      INTEGER(2) GETTEXTCOLOR
   END FUNCTION
END INTERFACE
```

### Description

GETTEXTCOLOR returns the text color index set by SETTEXTCOLOR. SETTEXTCOLOR affects text output with OUTTEXT, WRITE, and PRINT. The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. To access all system colors, use SETTEXTCOLORRGB, SETBKCOLORRGB, and SETCOLORRGB.

The default text color index is 15, which is associated with white unless the user remaps the palette.

### Output

The result type is INTEGER(2). It is the current text color index.

# GETTEXTPOSITION

*Returns the current text position.*

### Prototype

```
INTERFACE
  SUBROUTINE GETTEXTPOSITION(S)
    STRUCTURE /RCCOORD/
        INTEGER(2) ROW, COL
    END STRUCTURE
    RECORD /RCCOORD/ S
  END SUBROUTINE
END INTERFACE
```

*s*              Output. Derived type RCCOORD. Current text position.

```
TYPE RCCOORD
  INTEGER(2) ROW   ! Row coordinate
  INTEGER(2) COL   ! Column coordinate
END TYPE RCCOORD
```

### Description

The text position given by coordinates (1, 1) is defined as the upper-left corner of the text window. Text output from the OUTTEXT function (and WRITE and PRINT statements) begins at the current text position. Font text is not affected by the current text position. Graphics output, including OUTGTEXT output, begins at the current graphics output position, which is a separate position returned by GETCURRENTPOSITION.

# GETTEXTWINDOW

*Finds the boundaries of the current text window.*

### Prototype

```
INTERFACE
   SUBROUTINE gettextwindow(R1,C1,R2,C2)
      INTEGER(2) R1,C1,R2,C2
   END SUBROUTINE
END INTERFACE
```

R1, C1          Output. `INTEGER(2)`. Row and column coordinates for upper-left corner of the text window.

R2, C2          Output . `INTEGER(2)`. Row and column coordinates for lower-right corner of the text window.

### Description

Output from `OUTTEXT` and `WRITE` is limited to the text window. By default, this is the entire window, unless the text window is redefined by `SETTEXTWINDOW`.

The window defined by `SETTEXTWINDOW` has no effect on output from `OUTGTEXT`.

# GETVIEWCOORD

*Translates physical coordinates or window coordinates to viewport coordinates.*

### Prototype

```
INTERFACE GETVIEWCOORD
 SUBROUTINE GETVIEWCOORD(X,Y,S)
 INTEGER(2) X, Y
 STRUCTURE /XYCOORD/
      INTEGER(2) XCOORD, YCOORD
 END STRUCTURE
 RECORD /XYCOORD/ S
 END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| X, Y | Input. `INTEGER(2)`. Physical coordinates to be converted to viewport coordinates. |
| S | Output. Derived type xycoord. Viewport coordinates. |

```
TYPE XYCOORD
  INTEGER(2) XCOORD   ! x-coordinate
  INTEGER(2) YCOORD   ! y-coordinate
END TYPE XYCOORD
```

### Description

Viewport coordinates refer to an area of the screen defined as the viewport with `SETVIEWPORT`. Physical coordinates refer to the whole screen. Both take integer coordinate values. Window coordinates refer to a window sized with `SETWINDOW` or `SETWSIZEQQ`. Window coordinates are floating-point values and allow easy scaling of data to the window area.

# GETVIEWCOORD_W

*Translates physical coordinates or window coordinates to viewport coordinates.*

### Prototype

```
INTERFACE
    SUBROUTINE GETVIEWCOORD_W(WX,WY,S)
        DOUBLE PRECISION WX,WY
        STRUCTURE /XYCOORD/
            INTEGER(2) XCOORD, YCOORD
        END STRUCTURE
        RECORD /XYCOORD/ S
    END SUBROUTINE
END INTERFACE
```

WX, WY       Input. `REAL(8)`. Window coordinates to be converted to viewport coordinates.

S       Output. Derived type xycoord. Viewport coordinates.

```
TYPE XYCOORD
  INTEGER(2) XCOORD   ! x-coordinate
  INTEGER(2) ycYCOORDoord   ! y-coordinate
END TYPE XYCOORD
```

### Description

Viewport coordinates refer to an area of the screen defined as the viewport with `SETVIEWPORT`. Physical coordinates refer to the whole screen. Both take integer coordinate values. Window coordinates refer to a window sized with `SETWINDOW` or `SETWSIZEQQ`. Window coordinates are floating-point values and allow easy scaling of data to the window area.

# GETWINDOWCOORD

*Translates viewport coordinates to window coordinates.*

### Prototype

```
INTERFACE GETWINDOWCOORD
  SUBROUTINE GETWINDOWCOORD(X,Y,S)
   INTEGER(2) X, Y
   STRUCTURE /XYCOORD/
      DOUBLE PRECISION X, Y
   END STRUCTURE
   RECORD /wxycoord/ s
  END SUBROUTINE
 END INTERFACE
```

X, Y      Input. `INTEGER(2)`. Viewport coordinates to be converted to window coordinates.

WT      Output. Derived type `XYCOORD`. Window coordinates.

```
TYPE XYCOORD
  REAL(8) X   ! x-coordinate
  REAL(8) Y   ! y-coordinate
END TYPE XYCOORD
```

### Description

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with `SETVIEWPORT`. Both take integer coordinate values. Window coordinates refer to a window sized with `SETWINDOW` or `SETWSIZEQQ`. Window coordinates are floating-point values and allow easy scaling of data to the window area.

# GETWRITEMODE

*Returns the current logical write mode.*

### Prototype

```
INTERFACE
  FUNCTION GETWRITEMODE()
    INTEGER(2) GETWRITEMODE
  END FUNCTION
END INTERFACE
```

### Description

Returns the current logical write mode, which is used when drawing lines with the LINETO**,** POLYGON, and RECTANGLE functions. The write mode is set with SETWRITEMODE.

### Output

The result type is INTEGER(2). The result is the current write mode. The default value is $GPSET. Possible return values are:

| | |
|---|---|
| $GPSET | Causes lines to be drawn in the current graphics color. (default) |
| $GAND | Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color. |
| $GOR | Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color. |
| $GPRESET | Causes lines to be drawn in the color that is the logical NOT of the current graphics color. |
| $GXOR | Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color. |

# GRSTATUS

*Returns the status of the most recently
used graphics routine.*

### Prototype

```
INTERFACE
  FUNCTION GRSTATUS()
    integer*2 GRSTATUS
  END FUNCTION
END INTERFACE
```

### Description

Use GRSTATUS immediately following a call to a graphics routine to
determine if errors or warnings were generated. Return values less than 0
are errors, and values greater than 0 are warnings.

| | |
|---|---|
| $GRFILEWRITEERROR | Error writing bitmap file |
| $GRFILEOPENERROR | Error opening bitmap file |
| $GRIMAGEREADERROR | Error reading image |
| $GRBITMAPDISPLAYERROR | Error displaying bitmap |
| $GRBITMAPTOOLARGE | Bitmap too large |
| $GRIMPROPERBITMAPFORMAT | |
| | Improper format for bitmap file |
| $GRFILEREADERROR | Error reading file |
| $GRNOBITMAPFILE | No bitmap file |
| $GRINVALIDIMAGEBUFFER | Image buffer data inconsistent |
| $GRINSUFFICIENTMEMORY | Not enough memory to allocate buffer or to complete a fill operation |
| $GRINVALIDPARAMETER | One or more parameters invalid |
| $GRMODENOTSUPPORTED | Requested video mode not supported |
| $GRERROR | Graphics error |

| | |
|---|---|
| `$GROK` | Success |
| `$GRNOOUTPUT` | No action taken |
| `$GRCLIPPED` | Output was clipped to viewport |
| `$GRPARAMETERALTERED` | One or more input parameters was altered to be within range, or pairs of parameters were interchanged to be in the proper order |

After a graphics call, compare the return value of `GRSTATUS` to `$GROK` to determine if an error has occurred.

### Output

The result type is `INTEGER(2)`. The result is the status of the most recently used graphics function.

# IMAGESIZE

*Returns the number of bytes needed to store the image inside the specified bounding rectangle.*

### Prototype

```
INTERFACE
  FUNCTION IMAGESIZE(X1,Y1,X2,Y2)
     INTEGER(4) IMAGESIZE
     INTEGER(2) X1,Y1,X2,Y2
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| `X1, Y1` | InpuT. `INTEGER(2)`. Viewport coordinates for upper-left corner of image. |
| `X2, Y2` | Input. `INTEGER(2)`. Viewport coordinates for lower-right corner of image. |

### Description

IMAGESIZE defines the bounding rectangle in viewport-coordinate points (X1, Y1) and (X2, Y2). Returns the number of bytes needed to store the image inside the specified bounding rectangle. Useful for determining how much memory is needed for a call to GETIMAGE.

### Output

The result type is INTEGER(4). The result is the storage size of an image in bytes.

# IMAGESIZE_W

*Returns the number of bytes needed to store the image inside the specified bounding rectangle.*

### Prototype

```
INTERFACE
  FUNCTION IMAGESIZE_W(WX1,WY1,WX2,WY2)
    INTEGER(4) IMAGESIZE_W
    DOUBLE PRECISION WX1,WY1,WX2,WY2
  END FUNCTION
END INTERFACE
```

WX1, WY1          Input. REAL(8). Window coordinates for upper-left corner of image.

WX2, WY2          Input. REAL(8). Window coordinates for lower-right corner of image.

### Description

IMAGESIZE_W defines the bounding rectangle in terms of window-coordinate points (WX1, WY1) and (WX2, WY2).

The function returns the number of bytes needed to store the image inside the specified bounding rectangle. IMAGESIZE is useful for determining how much memory is needed for a call to GETIMAGE.

### Output

The result type is INTEGER(4). The result is the storage size of an image in bytes.

## LINETO

*Draws a line from the current graphics position up to and including the end point.*

### Prototype

```
INTERFACE
    FUNCTION LINETO(X,Y)
        INTEGER(2) LINETO,X,Y
    END FUNCTION
END INTERFACE
```

X, Y         Input. INTEGER(2). Viewport coordinates of end point.

### Description

The line is drawn using the current graphics color, logical write mode, and line style. The graphics color is set with SETCOLORRGB, the write mode with SETWRITEMODE, and the line style with SETLINESTYLE.

If no error occurs, LINETO sets the current graphics position to the viewport point (X, Y).

If you use FLOODFILLRGB to fill in a closed figure drawn with LINETO, the figure must be drawn with a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

### Output

The result type is `INTEGER(2)`. The result is a nonzero value if successful; otherwise, 0.

# LINETO_W

*Draws a line from the current graphics position up to and including the end point.*

### Prototype

```
INTERFACE
  FUNCTION LINETO_W(WX,WY)
    INTEGER(2) LINETO_W
    dDOUBLE PRECISION WX,WY
  END FUNCTION
END INTERFACE
```

WX, WY          Input. `REAL(8)`. Window coordinates of end point.

### Description

The line is drawn using the current graphics color, logical write mode, and line style. The graphics color is set with `SETCOLORRGB`, the write mode with `SETWRITEMODE`, and the line style with `SETLINESTYLE`.

If no error occurs, `LINETO_W` sets the current graphics position to the window point (`WX,WY`).

If you use `FLOODFILLRGB` to fill in a closed figure drawn with `LINETO_W`, the figure must be drawn with a solid line style. Line style is solid by default and can be changed with `SETLINESTYLE`.

### Output

The result type is `INTEGER(2)`. The result is a nonzero value if successful; otherwise, 0.

# LOADIMAGE

*Reads an image from a Windows bitmap
file and displays it at a specified
location.*

### Prototype

```
INTERFACE
  FUNCTION LOADIMAGE(FNAME, X, Y)
     INTEGER(4) LOADIMAGE, X, Y
     CHARACTER(LEN=*) FNAME
     END FUNCTION
  END INTERFACE
```

FNAME           Input. CHARACTER(LEN=*). Path of the bitmap file.

X, Y            Input. INTEGER(4). Viewport coordinates for
                upper-left corner of image display.

### Description

The image is displayed with the colors in the bitmap file. If the color palette
in the bitmap file is different from the current system palette, the current
palette is discarded and the bitmap's palette is loaded.

LOADIMAGE specifes the screen placement of the image in viewport
coordinates.

### Output

The result type is INTEGER(4). The result is zero if successful; otherwise,
a negative value.

# LOADIMAGE_W

*Reads an image from a Windows bitmap file and displays it at a specified location.*

### Prototype

```
INTERFACE
   FUNCTION LOADIMAGE_W(FNAME,WX,WY)
    INTEGER(4) LOADIMAGE_W
    CHARACTER(LEN=*) FNAME
    DOUBLE PRECISION WX, WY
  END FUNCTION
END INTERFACE
```

FNAME         Input. `CHARACTER(LEN=*)`. Path of the bitmap file.

WX ,WY        Input. `REAL(8)`. Window coordinates for upper-left corner of image display.

### Description

The image is displayed with the colors in the bitmap file. If the color palette in the bitmap file is different from the current system palette, the current palette is discarded and the bitmap's palette is loaded.

`LOADIMAGE_W` specifies the screen placement of the image in window coordinates.

### Output

The result type is `INTEGER(4)`. The result is zero if successful; otherwise, a negative value.

# MOVETO

*Moves the current graphics position to a
specified point. No drawing occurs.*

## Prototype

```
INTERFACE MOVETO
 SUBROUTINE MOVETO(X,Y,S)
  INTEGER(2) X, Y
  integer*2 y
  STRUCTURE /XYCOORD/
       INTEGER(2) XCOORD, YCOORD
       END STRUCTURE
 RECORD /XYCOORD/S
 END SUBROUTINE
END INTERFACE
```

X, Y          Input. `INTEGER(2)`. Viewport coordinates of the new
              graphics position.

S             Output. Derived type `XYCOORD`. Viewport coordinates
              of the previous graphics position.

```
TYPE XYCOORD
   INTEGER(2) XCOORD  ! x coordinate
   INTEGER(2) YCOORD  ! y coordinate
 END TYPE XYCOORD
```

## Description

`MOVETO` sets the current graphics position to the viewport coordinate `(X,Y)`.
It assign the coordinates of the previous position to `S` respectively.

# MOVETO_W

*Moves the current graphics position to a*
*specified point. No drawing occurs.*

## Prototype

```
INTERFACE
  SUBROUTINE MOVETO_W(WX,WY,S)
    DOUBLE PRECISION WX,WY
    STRUCTURE /WXYCOORD/
        DOUBLE PRECISION WX, WY
    END STRUCTURE
    RECORD /WXYCOORD/s
  END SUBROUTINE
END INTERFACE
```

WX,WY           Input. `REAL(8)`. Window coordinates of the new graphics position.

S           Output. Derived type `WXYCOORD`. Window coordinates of the previous graphics position.

```
TYPE WXYCOORD
   REAL(8) WX  ! x window coordinate
   REAL(8) WY  ! y window coordinate
 END TYPE WXYCOORD
```

## Description

`MOVETO_W` sets the current graphics position to the window coordinates
(`WX, WY`). Next call to `MOVETO_W` assigns the coordinates of the previous
position to `S`, respectively.

## OUTTEXT

*In text or graphics mode, sends a string
of text to the screen, including any
trailing blanks.*

### Prototype

```
INTERFACE
  SUBROUTINE OUTTEXT(TEXT)
    CHARACTER(LEN=*) TEXT
  END SUBROUTINE
END INTERFACE
```

TEXT            Input. `CHARACTER(LEN=*)`. String to be displayed.

### Description

Text output begins at the current text position in the color set with
`SETTEXTCOLORRGB` or `SETTEXTCOLOR`. No formatting is provided. After
it outputs the text, `OUTTEXT` updates the current text position.

## PIE

*Draws a pie-shaped wedge in the
current graphics color.*

### Prototype

```
INTERFACE
  FUNCTION PIE(I,X1,Y1,X2,Y2,X3,Y3,X4,Y4)
    INTEGER(2) PIE,I,X1,Y1,X2,Y2,X3,Y3,X4,Y4
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| I | Input. `INTEGER(2)`. Fill flag. One of the following symbolic constants: |
| | `$GFILLINTERIOR`  Fills the figure using the current color and fill mask. |
| | `$GBORDER`  Does not fill the figure. |
| X1,Y1 | Input. `INTEGER(2)`. Viewport coordinates for upper-left corner of bounding rectangle. |
| X2,Y2 | Input. `INTEGER(2)`. Viewport coordinates for lower-right corner of bounding rectangle. |
| X3,Y3 | Input. `INTEGER(2)`. Viewport coordinates of start vector. |
| X4,Y4 | Input. `INTEGER(2)`. Viewport coordinates of end vector. |

### Description

The border of the pie wedge is drawn in the current color set by `SETCOLORRGB`.

The `PIE` function uses the viewport-coordinate system. The center of the arc is the center of the bounding rectangle, which is specified by the viewport-coordinate points (`X1,Y1`) and (`X2,Y2`). The arc starts where it intersects an imaginary line extending from the center of the arc through (`X3,Y3`). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (`X4,Y4`).

The fill flag option `$GFILLINTERIOR` is equivalent to a subsequent call to `FLOODFILLRGB` using the center of the pie as the starting point and the current graphics color (set by `SETCOLORRGB`) as the fill color. If you want a fill color different from the boundary color, you cannot use the $GFILLINTERIOR option. Instead, after you have drawn the pie wedge, change the current color with `SETCOLORRGB` and then call `FLOODFILLRGB`. You must supply `FLOODFILLRGB` with an interior point in the figure you want to fill. You can get this point for the last drawn pie or arc by calling `GETARCINFO`.

If you fill the pie with FLOODFILLRGB, the pie must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

### Output

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the pie is clipped or partially out of bounds, the pie is considered successfully drawn and the return is 1. If the pie is drawn completely out of bounds, the return is 0.

## PIE_W

*Draws a pie-shaped wedge in the current graphics color.*

### Prototype

```
INTERFACE
  FUNCTION PIE_W(I,WX1,WY1,WX2,WY2,WX3,WY3,WX4,WY4)
    INTEGER(2) PIE_W, I
    DOUBLE PRECISION WX1,WY1,WX2,WY2,WX3,WY3,WX4,WY4
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| I | Input. INTEGER(2). Fill flag. One of the following symbolic constants: |
| | $GFILLINTERIOR Fills the figure using the current color and fill mask. |
| | $GBORDER Does not fill the figure. |
| WX1,WY1 | Input. REAL(8). Window coordinates for upper-left corner of bounding rectangle. |
| WX2,WY2 | Input. REAL(8). Window coordinates for lower-right corner of bounding rectangle. |

| WX3,WY3 | Input. REAL(8). Window coordinates of start vector. |
|---|---|
| WX4,WY4 | Input. REAL(8). Window coordinates of end vector. |

### Description

The border of the pie wedge is drawn in the current color set by SETCOLORRGB.

The PIE_W function uses the window-coordinate system. The center of the arc is the center of the bounding rectangle specified by the window-coordinate points (WX1,WY1) and (WX2,WY2). The arc starts where it intersects an imaginary line extending from the center of the arc through (WX3,WY3). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (WX4,WY4).

The fill flag option $GFILLINTERIOR is equivalent to a subsequent call to FLOODFILLRGB using the center of the pie as the starting point and the current graphics color (set by SETCOLORRGB) as the fill color. If you want a fill color different from the boundary color, you cannot use the $GFILLINTERIOR option. Instead, after you have drawn the pie wedge, change the current color with SETCOLORRGB and then call FLOODFILLRGB. You must supply FLOODFILLRGB with an interior point in the figure you want to fill. You can get this point for the last drawn pie or arc by calling GETARCINFO.

If you fill the pie with FLOODFILLRGB, the pie must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

### Output

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the pie is clipped or partially out of bounds, the pie is considered successfully drawn and the return is 1. If the pie is drawn completely out of bounds, the return is 0.

# POLYGON

*Draws a polygon using the current graphics color, logical write mode, and line style.*

### Prototype

```
INTERFACE
  FUNCTION POLYGON(CONTROL,LPPOINTS,CPOINTS)
    INTEGR(2) POLYGON,CONTROL,CPOINTS
        STRUCTURE /XYCOORD/
            INTEGER(2) XCOORD
            INTEGER(2) YCOORD
        END STRUCTURE
        RECORD /XYCOORD/LPPOINTS(*)
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| CONTROL | Input. INTEGER(2). Fill flag. One of the following symbolic constants: |

$GFILLINTERIOR Fills the figure using the current color and fill mask.

$GBORDER Does not fill the figure.

| | |
|---|---|
| LPPOINTS | Input. Derived type XYCOORD. Array of derived types defining the polygon vertices in viewport coordinates. |

```
TYPE XYCOORD
   INTEGER(2) XCOORD
   INTEGER(2) YCOORD
END TYPE XYCOORD
```

| | |
|---|---|
| CPOINTS | Input. INTEGER(2). Number of polygon vertices. |

### Description

The border of the polygon is drawn in the current graphics color, logical write mode, and line style, set with `SETCOLORRGB`, `SETWRITEMODE`, and `SETLINESTYLE`, respectively. The `POLYGON` routine uses the viewport-coordinate system (expressed in xycoord derived types).

The arguments `LPPOINTS` are arrays whose elements are `XYCOORD` or `WXYCOORD` derived types. Each element specifies one of the polygon's vertices. The argument `CPOINTS` is the number of elements (the number of vertices) in the `LPPOINTS` array.

Note that `POLYGON` draws between the vertices in their order in the array. Therefore, when drawing outlines, skeletal figures, or any other figure that is not filled, you need to be careful about the order of the vertices. If you don't want lines between some vertices, you may need to repeat vertices to make the drawing backtrack and go to another vertex to avoid drawing across your figure. Also, `POLYGON` draws a line from the last specified vertex back to the first vertex.

If you fill the polygon using `FLOODFILLRGB`, the polygon must be bordered by a solid line style. Line style is solid by default and can be changed with `SETLINESTYLE`.

### Output

The result type is `INTEGER(2)`. The result is nonzero if anything is drawn; otherwise, 0.

## POLYGON_W

*Draws a polygon using the current graphics color, logical write mode, and line style.*

### Prototype

`INTERFACE`

```
              FUNCTION POLYGON_W(CONTROL,LPPOINTS,CPOINTS)
                INTEGR(2) POLYGON_W,CONTROL,CPOINTS
                    STRUCTURE /WXYCOORD/
                        DOUBLE PRECISION WX, WY
                    END STRUCTURE
                    RECORD /WXYCOORD/LPPOINTS(*)
              END FUNCTION
            END INTERFACE
```

CONTROL        Input. `INTEGER(2)`. Fill flag. One of the following symbolic constants:

           `$GFILLINTERIOR` Fills the figure using the current color and fill mask.

           `$GBORDER`           Does not fill the figure.

LPPOINTS       Input. Derived type `WXYCOORD`. Array of derived types defining the polygon vertices in window coordinates.

```
                 TYPE WXYCOORD
                    REAL(W) WX, WY
                 END TYPE WXYCOORD
```

CPOINTS        Input. `INTEGER(2)`. Number of polygon vertices.

### Description

The border of the polygon is drawn in the current graphics color, logical write mode, and line style, set with `SETCOLORRGB`, `SETWRITEMODE`, and `SETLINESTYLE`, respectively. The `POLYGON_W` routine uses real-valued window coordinates (expressed in wxycoord types).

The arguments `LPPOINTS` are arrays whose elements are `XYCOORD` or `WXYCOORD` derived types. Each element specifies one of the polygon's vertices. The argument `CPOINTS` is the number of elements (the number of vertices) in the `LPPOINTS` array.

Note that `POLYGON_W` draws between the vertices in their order in the array. Therefore, when drawing outlines, skeletal figures, or any other figure that is not filled, you need to be careful about the order of the vertices. If you don't want lines between some vertices, you may need to repeat vertices to

make the drawing backtrack and go to another vertex to avoid drawing across your figure. Also, `POLYGON_W` draws a line from the last specified vertex back to the first vertex.

If you fill the polygon using `FLOODFILLRGB`, the polygon must be bordered by a solid line style. Line style is solid by default and can be changed with `SETLINESTYLE`.

### Output

The result type is `INTEGER(2)`. The result is nonzero if anything is drawn; otherwise, 0.

# PUTIMAGE

*Transfers the image stored in memory to the screen.*

### Prototype

```
INTERFACE
  SUBROUTINE PUTIMAGE(X,Y,IMAGE,ACTION)
     INTEGER(2) X,Y,ACTION
     INTEGER(1) IMAGE(*)
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| `X,Y` | Input. `INTEGER(2)`. Viewport coordinates for upper-left corner of the image when placed on the screen. |
| `IMAGE` | Input. `INTEGER(1)`. Array of single-byte integers. Stored image buffer. |
| `ACTION` | Input. `INTEGER(2)`. Interaction of the stored image with the existing screen image. One of the following symbolic constants: |

$GAND            Forms a new screen display as the logical AND of the stored image and the existing screen display. Points that have the same color in both the existing screen image and the stored image remain the same color, while points that have different colors are joined by a logical AND.

$GOR             Superimposes the stored image onto the existing screen display. The resulting image is the logical OR of the image.

$GPRESET         Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when it was taken from the screen by GETIMAGE, producing a negative image.

$GPSET           Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen by GETIMAGE.

$GXOR            Causes points in the existing screen image to be inverted wherever a point exists in the stored image. This behavior is like that of a cursor. If you perform an exclusive OR of an image with the background twice, the background is restored unchanged. This allows you to move an object around without erasing the background. The $GXOR constant is a special mode often used for animation.

In addition, the following ternary raster operation constants can be used (described in the online documentation for the WIN32 API BitBlt):

$GSRCCOPY (same as $GPSET)

$GSRCPAINT (same as $GOR)

$GSRCAND (same as $GAND)

$GSRCINVERT (same as $GXOR)

$GSRCERASE

$GNOTSRCCOPY (same as $GPRESET)

$GNOTSRCERASE

$GMERGECOPY

$GMERGEPAINT

$GPATCOPY

$GPATPAINT

$GPATINVERT

$GDSTINVERT

$GBLACKNESS

$GWHITENESS

### Description

PUTIMAGE places the upper-left corner of the image at the viewport coordinates (X,Y).

# PUTIMAGE_W

*Transfers the image stored in memory to the screen.*

### Prototype

```
    INTERFACE
      SUBROUTINE PUTIMAGE_W(WX,WY,IMAGE,ACTION)
        DOUBLE PRECISION WX, WY
        INTEGER(1) IMAGE(*)
!MS$ ATTRIBUTES REFERENCE ::   IMAGE
        INTEGER(2) ACTION
      END SUBROUTINE
```

```
          END INTERFACE
```

| | |
|---|---|
| WX,WY | Input. REAL(8). REAL(8). Window coordinates for upper-left corner of the image when placed on the screen. |
| IMAGE | Input. INTEGER(1). Array of single-byte integers. Stored image buffer. |
| ACTION | Input. INTEGER(2). Interaction of the stored image with the existing screen image. One of the following symbolic constants: |

| | |
|---|---|
| $GAND | Forms a new screen display as the logical AND of the stored image and the existing screen display. Points that have the same color in both the existing screen image and the stored image remain the same color, while points that have different colors are joined by a logical AND. |
| $GOR | Superimposes the stored image onto the existing screen display. The resulting image is the logical OR of the image. |
| $GPRESET | Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when it was taken from the screen by GETIMAGE_W, producing a negative image. |
| $GPSET | Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen by GETIMAGE_W. |
| $GXOR | Causes points in the existing screen image to be inverted wherever a point exists in the stored image. This behavior is like that of a cursor. If you perform an exclusive OR of an image with the background twice, the background is restored unchanged. This allows you to |

move an object around without erasing the background. The `$GXOR` constant is a special mode often used for animation.

In addition, the following ternary raster operation constants can be used (described in the online documentation for the WIN32 API BitBlt):

`$GSRCCOPY` (same as `$GPSET`)

`$GSRCPAINT` (same as `$GOR`)

`$GSRCAND` (same as `$GAND`)

`$GSRCINVERT` (same as `$GXOR`)

`$GSRCERASE`

`$GNOTSRCCOPY` (same as `$GPRESET`)

`$GNOTSRCERASE`

`$GMERGECOPY`

`$GMERGEPAINT`

`$GPATCOPY`

`$GPATPAINT`

`$GPATINVERT`

`$GDSTINVERT`

`$GBLACKNESS`

`$GWHITENESS`

## Description

`PUTIMAGE_W` places the upper-left corner of the image at the window coordinates (`WX, WY`).

# RECTANGLE

*Draws a rectangle using the current graphics color, logical write mode, and line style.*

### Prototype

```
INTERFACE
   FUNCTION RECTANGLE(CONTROL,X1,Y1,X2,Y2)
      INTEGER(2) RECTANGLE
      INTEGER(2) CONTROL,X1,Y1,X2,Y2
   END FUNCTION
END INTERFACE
```

| | | |
|---|---|---|
| CONTROL | Input. INTEGER(2). Fill flag. One of the following symbolic constants: | |
| | $GFILLINTERIOR | Draws a solid figure using the current color and fill mask. |
| | $GBORDER | Draws the border of a rectangle using the current color and line style. |
| X1,Y1 | Input. INTEGER(2). Viewport coordinates for upper-left corner of rectangle. | |
| X2,Y2 | Input. INTEGER(2). Viewport coordinates for lower-right corner of rectangle. | |

### Description

The RECTANGLE function uses the viewport-coordinate system. The viewport coordinates (X1,Y1) and (X2,Y2) are the diagonally opposed corners of the rectangle.

SETCOLORRGB sets the current graphics color. SETFILLMASK sets the current fill mask. By default, filled graphic shapes are filled solid with the current color.

If you fill the rectangle using FLOODFILLRGB, the rectangle must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

### Output

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0.

# RECTANGLE_W

*Draws a rectangle using the current graphics color, logical write mode, and line style.*

### Prototype

```
INTERFACE
  FUNCTION RECTANGLE_W(CONTROL,WX1,WY1,WX2,WY2)
    INTEGER(2) RECTANGLE_W,CONTROL
    DOUBLE PRECISION WX1,WY1,WX2,WY2
  END FUNCTION
END INTERFACE
```

CONTROL          Input. INTEGER(2). Fill flag. One of the following symbolic constants:

$GFILLINTERIOR   Draws a solid figure using the current color and fill mask.

$GBORDER         Draws the border of a rectangle using the current color and line style.

WX1,WY1          Input. REAL(8). Window coordinates for upper-left corner of rectangle.

WX2,WY2          Input. REAL(8). Window coordinates for lower-right corner of rectangle.

### Description

The `RECTANGLE_W` function uses the window-coordinate system. The window coordinates (`WX1`,`WY1`) and (`WX2`,`WY2`) are the diagonally opposed corners of the rectangle.

`SETCOLORRGB` sets the current graphics color. `SETFILLMASK` sets the current fill mask. By default, filled graphic shapes are filled solid with the current color.

If you fill the rectangle using `FLOODFILLRGB`, the rectangle must be bordered by a solid line style. Line style is solid by default and can be changed with `SETLINESTYLE`.

### Output

The result type is `INTEGER(2)`. The result is nonzero if successful; otherwise, 0.

# REMAPALLPALETTERGB

*Remaps a set of Red-Green-Blue (RGB) color values to indexes recognized by the video hardware.*

### Prototype

```
INTERFACE
   FUNCTION REMAPALLPALETTERGB(COLORS)
      INTEGER(2) REMAPALLPALETTERGB
      INTEGER(4) COLORS(*)
   END FUNCTION
END INTERFACE
```

COLORS          Input. `INTEGER(4)`. Ordered array of RGB color values to be mapped in order to indexes. Must hold 0-255 elements.

### Description

The `REMAPALLPALETTERGB` function remaps all of the available color indexes simultaneously (up to 236; 20 indexes are reserved by the operating system). The `COLORS` argument points to an array of RGB color values. The default mapping between the first 16 indexes and color values is shown in the following table.

| Index | Color | Index | Color |
|-------|-------|-------|-------|
| 0 | $BLACK | 8 | $GRAY |
| 1 | $BLUE | 9 | $LIGHTBLUE |
| 2 | $GREEN | 10 | $LIGHTGREEN |
| 3 | $CYAN | 11 | $LIGHTCYAN |
| 4 | $RED | 12 | $LIGHTRED |
| 5 | $MAGENTA | 13 | $LIGHTMAGENTA |
| 6 | $BROWN | 14 | $YELLOW |
| 7 | $WHITE | 15 | $BRIGHTWHITE |

The number of colors mapped can be fewer than 236 if the number of colors supported by the current video mode is fewer, but at most 236 colors can be mapped by `REMAPALLPALETTERGB`. Most Windows graphics drivers support a palette of 256K colors or more, of which only a few can be mapped into the 236 palette indexes at a time. To access and use all colors on the system, bypass the palette and use direct RGB color functions such as such as `SETCOLORRGB` and `SETPIXELSRGB`.

In each RGB color value, each of the three colors, red, green and blue, is represented by an eight-bit value (2 hex digits). In the values you specify with `REMAPALLPALETTERGB`, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 11111111 (hex FF) the maximum for each of the three components. For example, #008080 yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

### Output

The result type is `INTEGER(4)`. `REMAPALLPALETTERGB` returns 0 if successful; otherwise, - 1.

# REMAPPALETTERGB

*Remaps one color index to an RGB color value.*

### Prototype

```
INTERFACE
  FUNCTION REMAPPALETTERGB(INDEX,COLOR)
     INTEGER(4) REMAPPALETTERGB,COLOR
     INTEGER(2) INDEX
  END FUNCTION
END INTERFACE
```

COLOR       Input. `INTEGER(4)`. RGB color value to assign to a color index.

INDEX       Input. `INTEGER(2)`. Color index to be reassigned an RGB color.

### Description

The `REMAPPALETTERGB` function remaps one of the available color indexes (up to 236; 20 indexes are reserved by the operating system). The `COLOR` argument is the RGB color value to assign. The default mapping between the first 16 indexes and color values is shown in the following table..

| Index | Color | Index | Color |
|-------|-------|-------|-------|
| 0 | $BLACK | 8 | $GRAY |
| 1 | $BLUE | 9 | $LIGHTBLUE |

| Index | Color | Index | Color |
|-------|-------|-------|-------|
| 2 | $GREEN | 10 | $LIGHTGREEN |
| 3 | $CYAN | 11 | $LIGHTCYAN |
| 4 | $RED | 12 | $LIGHTRED |
| 5 | $MAGENTA | 13 | $LIGHTMAGENTA |
| 6 | $BROWN | 14 | $YELLOW |
| 7 | $WHITE | 15 | $BRIGHTWHITE |

The number of colors mapped can be fewer than 236 if the number of colors supported by the current video mode is fewer, but at most 236 colors can be mapped by REMAPPALETTERGB. Most Windows graphics drivers support a palette of 256K colors or more, of which only a few can be mapped into the 236 palette indexes at a time. To access and use all colors on the system, bypass the palette and use direct RGB color functions such as such as SETCOLORRGB and SETPIXELSRGB.

In each RGB color value, each of the three colors, red, green and blue, is represented by an eight-bit value (2 hex digits). In the values you specify with REMAPPALETTERGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 11111111 (hex FF) the maximum for each of the three components. For example, #008080 yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

### Output

The result type is INTEGER(4). REMAPPALETTERGB returns the previous color assigned to the index.

# SAVEIMAGE

*Saves an image from a specified portion of the screen into a Windows bitmap file.*

### Prototype

```
INTERFACE
    INTEGER(4) FUNCTION SAVEIMAGE(FNAME,X1,Y1,X2,Y2)
            CHARACTER(LEN=*) FNAME
            INTEGER(4) X1,Y1,X2,Y2
        END FUNCTION
    END INTERFACE
```

FNAME           Input. CHARACTER(LEN=*). Path of the bitmap file.

X1,Y1           Input. INTEGER(4). Viewport coordinates for upper-left corner of the screen image to be captured.

X2,Y2           Input. INTEGER(4). Viewport coordinates for lower-right corner of the screen image to be captured.

### Description

The SAVEIMAGE function captures the screen image within a rectangle defined by the upper-left and lower-right screen coordinates and stores the image as a Windows bitmap file specified by FNAME. The image is stored with a palette containing the colors displayed on the screen.

SAVEIMAGE defines the bounding rectangle in viewport coordinates.

### Output

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

# SAVEIMAGE_W

*Saves an image from a specified portion of the screen into a Windows bitmap file.*

### Prototype

```
INTERFACE
    INTEGER(4) FUNCTION SAVEIMAGE_W(FNAME,X1,Y1,X2,Y2)
            CHARACTER(LEN=*) FNAME
            DOUBLE PRECISION WX1,WY1,WX2,WY2
        END FUNCTION
      END INTERFACE
```

FNAME           Input. `CHARACTER(LEN=*)`. Path of the bitmap file.

X1,Y1           Input. `REAL(8)`. Window coordinates for upper-left corner of the screen image to be captured.

X2,Y2           Input. `REAL(8)`. Window coordinates for lower-right corner of the screen image to be captured.

### Description

The `SAVEIMAGE_W` function captures the screen image within a rectangle defined by the upper-left and lower-right screen coordinates and stores the image as a Windows bitmap file specified by `FNAME`. The image is stored with a palette containing the colors displayed on the screen.

`SAVEIMAGE_W` defines the bounding rectangle in window coordinates.

### Output

The result type is `INTEGER(4)`. The result is zero if successful; otherwise, a negative value.

# SCROLLTEXTWINDOW

*Scrolls the contents of a text window.*

### Prototype

```
INTERFACE
  SUBROUTINE SCROLLTEXTWINDOW(ROWS)
    INTEGER(2) ROWS
  END SUBROUTINE
END INTERFACE
```

ROWS            Input. INTEGER(2). Number of rows to scroll.

### Description

The SCROLLTEXTWINDOW subroutine scrolls the text in a text window (previously defined by SETTEXTWINDOW). The default text window is the entire window.

The *rows* argument specifies the number of lines to scroll. A positive value for *rows* scrolls the window up (the usual direction); a negative value scrolls the window down. Specifying a number larger than the height of the current text window is equivalent to calling CLEARSCREEN ($GWINDOW). A value of 0 for ROWS has no effect.

# SETBKCOLOR

*Sets the current background color index for both text and graphics.*

### Prototype

```
INTERFACE
  FUNCTION SETBKCOLOR(COLOR)
```

```
            INTEGER(4) SETBKCOLOR,COLOR
         END FUNCTION
      END INTERFACE
```

COLOR          Input. `INTEGER(4)`. Color index to set the background color to.

### Description

`SETBKCOLOR` changes the background color index for both text and graphics. The color index of text over the background color is set with `SETTEXTCOLOR`. The color index of graphics over the background color (used by drawing functions such as `FLOODFILL` and `ELLIPSE`) is set with `SETCOLOR`. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use `SETBKCOLORRGB`, `SETCOLORRGB`, and `SETTEXTCOLORRGB`.

Changing the background color index does not change the screen immediately. The change becomes effective when `CLEARSCREEN` is executed or when doing text input or output, such as with `READ`, `WRITE`, or `OUTTEXT`. The graphics output function `OUTGTEXT` does not affect the color of the background.

Generally, `INTEGER(4)` color arguments refer to color values and `INTEGER(2)` color arguments refer to color indexes. The two exceptions are `GETBKCOLOR` and `SETBKCOLOR`. The default background color index is 0, which is associated with black unless the user remaps the palette with `REMAPPALETTERGB`.

### Output

The result type is `INTEGER(4)`. The result is the previous background color index.

# SETCLIPRGN

*Limits graphics output to part of the
screen.*

### Prototype

```
INTERFACE
   SUBROUTINE SETCLIPRGN(X1,Y1,X2,Y2)
      INTEGER(2) X1,Y1,X2,Y2
   END SUBROUTINE
END INTERFACE
```

X1,Y1          Input. INTEGER(2). Physical coordinates for upper-left
               corner of clipping region.

X2,Y2          Input. INTEGER(2). Physical coordinates for
               lower-right corner of clipping region.

### Description

The SETCLIPRGN function limits the display of subsequent graphics output
and font text output to that which fits within a designated area of the screen
(the "clipping region"). The physical coordinates (X1,Y1) and (X2,Y2) are
the upper-left and lower-right corners of the rectangle that defines the
clipping region. The SETCLIPRGN function does not change the
viewport-coordinate system; it merely masks graphics output to the screen.

SETCLIPRGN affects graphics and font text output only, such as
OUTGTEXT. To mask the screen for text output using OUTTEXT, use
SETTEXTWINDOW.

# SETCOLOR

*Sets the current graphics color index.*

### Prototype

```
INTERFACE
  FUNCTION SETCOLOR(COLOR)
    INTEGER(2) SETCOLOR
    INTEGER(2) COLOR
  END FUNCTION
END INTERFACE
```

COLOR          Input. INTEGER(2). Color index to set the current graphics color to.

### Description

The SETCOLOR function sets the current graphics color index, which is used by graphics functions such as ELLIPSE. The background color index is set with SETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB.

### Output

The result type is INTEGER(2). The result is the previous color index if successful; otherwise, -1.

# SETFILLMASK

*Sets the current fill mask to a new pattern.*

### Prototype

```
INTERFACE
  SUBROUTINE SETFILLMASK(MASK)
    INTEGER(1) MASK(8)
  END SUBROUTINE
END INTERFACE
```

MASK          Input. INTEGER(1). One-dimensional array of length 8.

### Description

There are 8 bytes in MASK, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of MASK becomes the top 8 bits of the pattern, and the eighth element (byte) of MASK becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of zero are set to the current background color. The current graphics color is set with SETCOLORRGB or SETCOLOR. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with SETFILLMASK), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (FLOODFILLRGB, PIE, ELLIPSE, POLYGON, and RECTANGLE).

To change the current fill mask, determine the array of bytes that corresponds to the desired bit pattern and set the pattern with SETFILLMASK, as in the following example.

# SETLINESTYLE

*Sets the current line style to a new line style.*

## Prototype

```
INTERFACE
  SUBROUTINE SETLINESTYLE(MASK)
    INTEGER(2) MASK
  END SUBROUTINE
END INTERFACE
```

MASK          Input. INTEGER(2). Desired Quickwin line-style mask. (See the table below.)

## Description

The mask is mapped to the style that most closly equivalences the the percentage of the bits in the mask that are set. The style produces lines that cover a certain percentage of the pixels in that line.

SETLINESTYLE sets the style used in drawing a line. You can choose from the following styles:

| QuickWin Mask | Internal Windows Style | Selection Criteria | Appearance |
|---|---|---|---|
| 0xFFFF | PS_SOLID | 16 bits on | _____ |
| 0xEEEE | PS_DASH | 11 to 15 bits on | --------------- |
| 0xECEC | PS_DASHDOT | 10 bits on | -.-.-.-.-.-.-.- |
| 0xECCC | PS_DASHDOTDOT | 9 bits on | -..-..-..-..-..-. |
| 0xAAAA | PS_DOT | 1 to 8 bits on | ...................... |
| 0x0000 | PS_NULL | 0 bits on | |

SETLINESTYLE affects the drawing of straight lines as in LINETO, POLYGON, and RECTANGLE, but not the drawing of curved lines as in ARC, ELLIPSE, or PIE.

The current graphics color is set with SETCOLORRGB or SETCOLOR. SETWRITEMODE affects how the line is displayed.

# SETPIXEL

*Sets a pixel at a specified location to the current graphics color index.*

### Prototype

```
INTERFACE
    FUNCTION SETPIXEL(X,Y)
        INTEGER(2) SETPIXEL,X,Y
    END FUNCTION
END INTERFACE
```

X,Y        Input. INTEGER(2). Viewport coordinates for target pixel.

### Description

SETPIXEL sets the specified pixel to the current graphics color index. The current graphics color index is set with SETCOLOR and retrieved with GETCOLOR. The non-RGB color functions (such as SETCOLOR and SETPIXELS) use color indexes rather than true color values.

### Output

The result type is INTEGER(2). The result is the previous color index of the target pixel if successful; otherwise, -1 (for example, if the pixel lies outside the clipping region).

# SETPIXEL_W

*Sets a pixel at a specified location to the current graphics color index.*

### Prototype

```
INTERFACE
  FUNCTION SETPIXEL_W(WX,WY)
    INTEGER(2) SETPIXEL_W
    DOUBLE PRECISION WX,WY
  END FUNCTION
END INTERFACE
```

WX,WY          Input. REAL(8). Window coordinates for target pixel.

### Description

SETPIXEL_W sets the specified pixel to the current graphics color index. The current graphics color index is set with SETCOLOR and retrieved with GETCOLOR. The non-RGB color functions (such as SETCOLOR and SETPIXELS) use color indexes rather than true color values.

### Output

The result type is INTEGER(2). The result is the previous color index of the target pixel if successful; otherwise, -1 (for example, if the pixel lies outside the clipping region).

# SETPIXELS

*Sets the color indexes of multiple pixels.*

### Prototype

```
INTERFACE
  SUBROUTINE SETPIXELS(N, X, Y, COLOR)
    INTEGER(4)N        ! size of arrays
    INTEGER(2)X(*),Y(*)! x, y coordinates
    INTEGER(2) COLOR(*)! palette indices
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| N | Input. INTEGER(4). Number of pixels to set. Sets the number of elements in the other arguments. |
| X,Y | Input. INTEGER(2). Parallel arrays containing viewport coordinates of pixels to set. |
| COLOR | Input. INTEGER(2). Array containing color indexes to set the pixels to. |

### Description

SETPIXELS sets the pixels specified in the arrays *x* and *y* to the color indexes in COLOR. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to SETPIXELS with N less than 1 are also ignored. SETPIXELS is a much faster way to set multiple pixel color indexes than individual calls to SETPIXEL.

Unlike SETPIXELS, SETPIXELSRGB gives access to the full color capacity of the system by using direct color values rather than indexes to a palette. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

# SETTEXTCOLOR

*Sets the current text color index.*

### Prototype

```
INTERFACE
  FUNCTION SETTEXTCOLOR(INDEX)
    INTEGER(2) SETTEXTCOLOR,INDEX
  END FUNCTION
END INTERFACE
```

INDEX            Input. INTEGER(2). Color index to set the text color to.

### Description

SETTEXTCOLOR sets the current text color index. The default value is 15, which is associated with white unless the user remaps the palette. GETTEXTCOLOR returns the text color index set by SETTEXTCOLOR. SETTEXTCOLOR affects text output with OUTTEXT, WRITE, and PRINT.

### Output

The result type is INTEGER(2). The result is the previous text color index.

---

# SETTEXTPOSITION

*Sets the current text position to a specified position relative to the current text window.*

---

## Prototype

```
INTERFACE
 SUBROUTINE SETTEXTPOSITION(ROW, COL, S)
    INTEGER(2) ROW, COL
    integer*2 col
    STRUCTURE /RCCOORD/
     INTEGER(2) ROW, COL
     integer*2 col
    END STRUCTURE
    RECORD /RCCOORD/s
 END SUBROUTINE
END INTERFACE
```

ROW             Input. `INTEGER(2)`. New text row position.

COL             Input. `INTEGER(2)`. New text column position.

S               Output. Derived type rccoord. Previous text position.

```
TYPE RCCOORD
   INTEGER(2) ROW ! Row coordinate
   INTEGER(2) COL ! Column coordinate
 END TYPE RCCOORD
```

## Description

Subsequent text output with the `OUTTEXT` function (as well as standard console I/O statements, such as `PRINT` and `WRITE`) begins at the point (`ROW`, `COL`).

# SETTEXTWINDOW

*Sets the current text window.*

### Prototype

```
INTERFACE
   SUBROUTINE SETTEXTWINDOW(R1,C1,R2,C2)
      iNTEGER(2) R1,C1,R2,C2
   END SUBROUTINE
END INTERFACE
```

R1,C1   Input. INTEGER(2). Row and column coordinates for upper-left corner of the text window.

R2,C2   Input. INTEGER(2). Row and column coordinates for lower-right corner of the text window.

### Description

SETTEXTWINDOW specifies a window in row and column coordinates where text output to the screen using OUTTEXT, WRITE, or PRINT will be displayed. You set the text location within this window with SETTEXTPOSITION.

Text is output from the top of the window down. When the window is full, successive lines overwrite the last line.

SETTEXTWINDOW does not affect the output of the graphics text routine OUTGTEXT. Use the SETVIEWPORT function to control the display area for graphics output.

# SETVIEWORG

*Moves the viewport-coordinate origin
(0, 0) to the specified physical point.*

## Prototype

```
INTERFACE
 SUBROUTINE SETVIEWORG(X,Y,S)
 INTEGER(2) X,Y
 integer*2 y
 STRUCTURE /XYCOORD/
      INTEGER(2) XCOORD,YCOORD
 END STRUCTURE
 RECORD /XYCOORD/s
 END SUBROUTINE
END INTERFACE
```

X,Y         Input. `INTEGER(2)`. Physical coordinates of new viewport origin.

S         Output. Derived type `XYCOORD`. Physical coordinates of the previous viewport origin.

```
TYPE XYCOORD
INTEGER(2) XCOORD ! x-coordinate
INTEGER(2) YCOORD ! y-coordinate
END TYPE XYCOORD
```

## Description

The `XYCOORD` type variable *s*, returns the physical coordinates of the previous viewport origin.

# SETVIEWPORT

*Redefines the graphics viewport*

### Prototype

```
INTERFACE
  SUBROUTINE SETVIEWPORT(X1,Y1,X2,Y2)
    INTEGER(2) X1,Y1,X2,Y2
  END SUBROUTINE
END INTERFACE
```

X1,Y1        Input. `INTEGER(2)`. Physical coordinates for upper-left corner of viewport.

X2,Y2        Input. `INTEGER(2)`. Physical coordinates for lower-right corner of viewport.

### Description

Redefines the graphics viewport by defining a clipping region in the same manner as `SETCLIPRGN` and then setting the viewport-coordinate origin to the upper-left corner of the region. The physical coordinates (`X1,Y1`) and (`X2,Y2`) are the upper-left and lower-right corners of the rectangular clipping region. Any window transformation done with the `SETWINDOW` function is relative to the viewport, not the entire screen.

# SETWINDOW

*Defines a window bound by the specified coordinates.*

### Prototype

```
INTERFACE
```

```
            FUNCTION SETWINDOW(FINVERT,WX1,WY1,WX2,WY2)
              INTEGER(2) SETWINDOW
              LOGICAL(2) FINVERT
              DOUBLE PRECISION WX1,WY1,WX2,WY
            END FUNCTION
          END INTERFACE
```

FINVERT         Input. LOGICAL(2). Direction of increase of the y-axis.
                If FINVERT is .TRUE., the y-axis increases from the
                window bottom to the window top (as Cartesian
                coordinates). If FINVERT is .FALSE., the y-axis
                increases from the window top to the window bottom
                (as pixel coordinates).

WX1,WY1         Input. REAL(8). Window coordinates for upper-left
                corner of window.

WX2,WY2         Input. REAL(8). Window coordinates for lower-right
                corner of window.

### Description

The SETWINDOW function determines the coordinate system used by all
window-relative graphics routines. Any graphics routines that end in _W
(such as ARC_W, RECTANGLE_W, and LINETO_W) use the coordinate sytem
set by SETWINDOW.

Any window transformation done with the SETWINDOW function is relative
to the viewport, not the entire screen.

An arc drawn using inverted window coordinates is not an upside-down
version of an arc drawn with the same parameters in a noninverted window.
The arc is still drawn counterclockwise, but the points that define where the
arc begins and ends are inverted.

If WX1 equals WX2 or WY1 equals WY2, SETWINDOW fails.

### Output

The result type is INTEGER(2). The result is The result is nonzero if
successful; otherwise, 0 (for example, if the program that calls SETWINDOW
is not in a graphics mode).

# SETWRITEMODE

*Sets the current logical write mode.*

```
Prototype
      INTERFACE
        FUNCTION SETWRITEMODE(WMODE)
           INTEGER(2) SETWRITEMODE,WMODE
        END FUNCTION
      END INTERFACE
```

WMODE      Input. `INTEGER(2)`. Write mode to be set. One of the following symbolic constants :

       `$GPSET`    Causes lines to be drawn in the current graphics color. (Default)

       `$GAND`    Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color.

       `$GOR`    Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color.

       `$GPRESET`   Causes lines to be drawn in the color that is the logical NOT of the current graphics color.

       `$GXOR`   Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color.

In addition, one of the following binary raster operation constants can be used (described in the online documentation for the WIN32 API SetROP2):

`$GR2_BLACK`

`$GR2_NOTMERGEPEN`

`$GR2_MASKNOTPEN`

$GR2_NOTCOPYPEN (same as $GPRESET)

$GR2_MASKPENNOT

$GR2_NOT

$GR2_XORPEN (same as $GXOR)

$GR2_NOTMASKPEN

$GR2_MASKPEN (same as $GAND)

$GR2_NOTXORPEN

$GR2_NOP

$GR2_MERGENOTPEN

$GR2_COPYPEN (same as $GPSET)

$GR2_MERGEPENNOT

$GR2_MERGEPEN (same as $GOR)

$GR2_WHITE

### Description

Sets the current logical write mode, which is used when drawing lines with the LINETO, POLYGON, and RECTANGLE functions. The current graphics color is set with SETCOLORRGB (or SETCOLOR) and the current background color is set with SETBKCOLORRGB (or SETBKCOLOR). As an example, suppose you set the background color to yellow (#00FFFF) and the graphics color to purple (#FF00FF) with the following commands:

```
OLDCOLOR = SETBKCOLORRGB(#00FFFF)
CALL CLEARSCREEN($GCLEARSCREEN)
OLDCOLOR = SETCOLORRGB(#FF00FF)
```

If you then set the write mode with the $GAND option, lines are drawn in red (#0000FF); with the $GOR option, lines are drawn in white (#FFFFFF); with the $GXOR option, lines are drawn in turquoise (#FFFF00); and with the $GPRESET option, lines are drawn in green (#00FF00). Setting the write mode to $GPSET causes lines to be drawn in the graphics color.

### Output

The result type is INTEGER(2). The result is the previous write mode if successful; otherwise, -1.

# WRAPON

*Controls the text output wrap.*

### Prototype

```
INTERFACE
  FUNCTION WRAPON(OPTION)
    INTEGER(2) WRAPON,OPTION
  END FUNCTION
END INTERFACE
```

OPTION          Input. INTEGER(2). Wrap mode. One of the following symbolic constants:

$GWRAPOFF   Truncates lines at right edge of window border.

$GWRAPON    Wraps lines at window border, scrolling if necessary.

### Description

Controls whether text output with the OUTTEXT function wraps to a new line or is truncated when the text output reaches the edge of the defined text window. WRAPON does not affect font routines such as OUTGTEXT.

### Output

The result type is INTEGER(2). The result is the previous value of OPTION.

## Per Pixel and Color Functions

---

# GETCOLORRGB

*Gets the current graphics color*
*Red-Green-Blue (RGB) value.*

### Prototype

```
INTERFACE
  FUNCTION GETCOLRRGB()
    integer*4 GETCOLRRGB
  END FUNCTION
END INTERFACE
```

### Description

Gets the current graphics color Red-Green-Blue (RGB) value (used by graphics functions such as ARC, ELLIPSE, and FLOODFILLRGB). In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETCOLORRGB returns the RGB color value of graphics over the background color (used by graphics functions such as ARC, ELLIPSE, and FLOODFILLRGB), set with SETCOLORRGB. GETBKCOLORRGB returns the RGB color value of the current background for both text and graphics, set

with SETBKCOLORRGB. GETTEXTCOLORRGB returns the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT), set with SETTEXTCOLORRGB.

SETCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). The result is the RGB value of the current graphics color.

# GETBKCOLORRGB

*Gets the current background*
*Red-Green-Blue (RGB) color value for*
*both text and graphics.*

### Prototype

```
INTERFACE
  FUNCTION GETBKCOLORRGB()
    INTEGER(4) GETBKCOLORRGB
  END FUNCTION
END INTERFACE
```

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETBKCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETBKCOLORRGB returns the RGB color value of the current background for both text and graphics, set with SETBKCOLORRGB. The RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT) is set with SETTEXTCOLORRGB and returned with GETTEXTCOLORRGB. The RGB color value of graphics over the background color (used by graphics functions such as ARC, OUTGTEXT, and FLOODFILLRGB) is set with SETCOLORRGB and returned with GETCOLORRGB.

SETBKCOLORRGB (and the other RGB color selection functions SETCOLORRGB and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETBKCOLOR, SETCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). The result is the RGB value of the current background color for both text and graphics.

# GETPIXELRGB

*Returns the Red-Green-Blue (RGB)
color value of the pixel at a specified
location.*

### Prototype

```
INTERFACE
  FUNCTION GETPIXELRGB(X,Y)
    INTEGER(4) GETPIXELRGB
    INTEGER(2) X,Y
  END FUNCTION
END INTERFACE
```

X,Y             Input. INTEGER(2). Viewport coordinates for pixel
                position.

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETPIXELRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETPIXELRGB returns the true color value of the pixel, set with SETPIXELRGB, SETCOLORRGB, SETBKCOLORRGB, or SETTEXTCOLORRGB, depending on the pixel's position and the current configuration of the screen.

SETPIXELRGB (and the other RGB color selection functions SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB) sets colors to a color value chosen from the entire available range. The non-RGB color

functions (SETPIXELS, SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). The result is the pixel's current RGB color value.

## GETPIXELRGB_W

*Returns the Red-Green-Blue (RGB) color value of the pixel at a specified location.*

### Prototype

```
INTERFACE
  FUNCTION GETPIXELRGB_W(WX,WY)
    INTEGER(4) GETPIXELRGB_W
    REAL*8          WX,WY
    REAL*8          wy
  END FUNCTION
END INTERFACE
```

WX,WY      Input. REAL(8). Window coordinates for pixel position.

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with `GETPIXELRGB_W`, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

`GETPIXELRGB_W` returns the true color value of the pixel, set with `SETPIXELRGB_W`, `SETCOLORRGB`, `SETBKCOLORRGB`, or `SETTEXTCOLORRGB`, depending on the pixel's position and the current configuration of the screen.

`SETPIXELRGB_W` (and the other RGB color selection functions `SETCOLORRGB`, `SETBKCOLORRGB`, and `SETTEXTCOLORRGB`) sets colors to a color value chosen from the entire available range. The non-RGB color functions (`SETPIXELS`, `SETCOLOR`, `SETBKCOLOR`, and `SETTEXTCOLOR`) use color indexes rather than true color values. If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is `INTEGER(4)`. The result is the pixel's current RGB color value.

# SETPIXELSRGB

*Sets multiple pixels to the given*
*Red-Green-Blue (RGB) color.*

### Prototype

```
INTERFACE
  SUBROUTINE SETPIXELSRGB(N,X,Y,COLOR)
    INTEGER(4) N
    INTEGER(2) X(*),Y(*)
    INTEGER(4) COLOR(*)
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| N | Input. INTEGER(4). Number of pixels to be changed. Determines the number of elements in arrays *x* and *y*. |
| X,Y | Input. INTEGER(2). Parallel arrays containing viewport coordinates of the pixels to set. |
| COLOR | Input. INTEGER(4). Array containing the RGB color values to set the pixels to. Range and result depend on the system's display adapter. |

### Description

SETPIXELSRGB sets the pixels specified in the arrays *x* and *y* to the RGB color values in COLOR. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you set with SETPIXELSRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

A good use for SETPIXELSRGB is as a buffering form of SETPIXELRGB, which can improve performance substantially. The example code shows how to do this.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to SETPIXELSRGB with *n* less than 1 are also ignored.

SETPIXELSRGB (and the other RGB color selection functions such as SETPIXELRGB and SETCOLORRGB) sets colors to values chosen from the entire available range. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

# GETPIXELSRGB

*Returns the Red-Green-Blue (RGB)*
*color values of multiple pixels.*

### Prototype

```
INTERFACE
  SUBROUTINE GETPIXELSRGB(N,X,Y,COLOR)
    INTEGER(4) N
    INTEGER(2) X(*),Y(*)
    INTEGER(4) COLOR (*)
  END SUBROUTINE
```

```
      END INTERFACE
```

N                  Input. INTEGER(4). Number of pixels to get. Sets the
                   number of elements in the other argument arrays.

X,Y                Input. INTEGER(2). Parallel arrays containing
                   viewport coordinates of pixels.

COLOR              Output. INTEGER(4). Array to be filled with RGB
                   color values of the pixels at *x* and *y*.

### Description

GETPIXELS fills in the array COLOR with the RGB color values of the
pixels specified by the two input arrays *x* and *y*. These arrays are parallel:
the first element in each of the three arrays refers to a single pixel, the
second element refers to the next pixel, and so on.

In each RGB color value, each of the three colors, red, green, and blue, is
represented by an eight-bit value (2 hex digits). In the value you retrieve
with GETPIXELSRGB, red is the rightmost byte, followed by green and
blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111
(hex FF) the maximum for each of the three components. For example,
#0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000
full-intensity blue, and #FFFFFF full-intensity for all three, resulting in
bright white.

GETPIXELSRGB is a much faster way to acquire multiple pixel RGB colors
than individual calls to GETPIXELRGB. GETPIXELSRGB returns an array
of true color values of multiple pixels, set with SETPIXELSRGB,
SETCOLORRGB, SETBKCOLORRGB, or SETTEXTCOLORRGB, depending on
the pixels' positions and the current configuration of the screen.

SETPIXELSRGB (and the other RGB color selection functions
SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB) sets colors
to a color value chosen from the entire available range. The non-RGB color
functions (SETPIXELS, SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR)
use color indexes rather than true color values. If you use color indexes, you
are limited to the colors available in the palette, at most 256. Some display
adapters (SVGA and true color) are capable of creating 262,144 (256K)

colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

# SETCOLORRGB

*Sets the current graphics color to the*
*specified Red-Green-Blue (RGB) value.*

### Prototype

```
INTERFACE
  FUNCTION SETCOLORRGB(COLOR)
    INTEGER(4) SETCOLORRGB,COLOR
  END FUNCTION
END INTERFACE
```

COLOR            Input. `INTEGER(4)`. RGB color value to set the current graphics color to. Range and result depend on the system's display adapter.

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with `SETCOLORRGB`, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

`SETCOLORRGB` sets the RGB color value of graphics over the background color, used by the following graphics functions: `ARC`, `ELLIPSE`, `FLOODFILL`, `LINETO`, `OUTGTEXT`, `PIE`, `POLYGON`, `RECTANGLE`, and

SETPIXEL. SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. SETTEXTCOLORRGB sets the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT).

SETCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB, and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). The result is the previous RGB color value.

# SETBKCOLORRGB

*Sets the current background color to the given Red-Green-Blue (RGB) value.*

### Prototype

```
INTERFACE
  FUNCTION SETBKCOLORRGB(color)
     INTEGER(4) SETBKCOLORRGB,COLOR
  END FUNCTION
END INTERFACE
```

COLOR            Input. INTEGER(4). RGB color value to set the background color to. Range and result depend on the system's display adapter.

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETBKCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. The RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT) is set with SETTEXTCOLORRGB. The RGB color value of graphics over the background color (used by graphics functions such as ARC, OUTGTEXT, and FLOODFILLRGB) is set with SETCOLORRGB.

SETBKCOLORRGB (and the other RGB color selection functions SETCOLORRGB, and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color

### Output

The result type is INTEGER(4). The result is the previous background RGB color value.

# SETPIXELRGB

*Sets a pixel at a specified location to the specified Red-Green-Blue (RGB) color value.*

### Prototype

```
INTERFACE
  FUNCTION SETPIXELRGB(X,Y,COLOR)
     INTEGER(4) SETPIXELRGB,COLOR
     INTEGER(2) X,Y
   END FUNCTION
END INTERFACE
```

| | |
|---|---|
| X,Y | Input. `INTEGER(2)`. Viewport coordinates for target pixel. |
| COLOR | Input. `INTEGER(4)`. RGB color value to set the pixel to. Range and result depend on the system's display adapter. |

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETPIXELRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

SETPIXELRGB (and the other RGB color selection functions such as SETPIXELSRGB, SETCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). The result is the previous RGB color value of the pixel.

# SETPIXELRGB_W

*Sets a pixel at a specified location to the specified Red-Green-Blue (RGB) color value.*

### Prototype

```
INTERFACE
  FUNCTION SETPIXELRGB_W(X,Y,COLOR)
    INTEGER(4) SETPIXELRGB_W,COLOR
    REAL*8        WX,WY
  END FUNCTION
END INTERFACE
```

WX,WY          Input. REAL(8). Window coordinates for target pixel.

COLOR          Input. INTEGER(4). RGB color value to set the pixel to. Range and result depend on the system's display adapter.

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETPIXELRGB_W, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

SETPIXELRGB_W (and the other RGB color selection functions such as SETPIXELSRGB, SETCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). The result is the previous RGB color value of the pixel.

# RGBTOINTEGER

*Converts three integers specifying red, green, and blue color intensities into a four-byte RGB integer.*

### Prototype

INTERFACE

```
FUNCTION RGBTOINTEGER(RED,GREEN,BLUE)
   INTEGER(4) RGBTOINTEGER,RED,GREEN,BLUE
 END FUNCTION
END INTERFACE
```

| | |
|---|---|
| RED | Input. INTEGER(4). Intensity of the red component of the RGB color value. Only the lower 8 bits of RED are used. |
| GREEN | Input. INTEGER(4). Intensity of the green component of the RGB color value. Only the lower 8 bits of GREEN are used. |
| BLUE | Input. INTEGER(4). Intensity of the blue component of the RGB color value. Only the lower 8 bits of BLUE are used. |

### Description

Converts three integers specifying red, green, and blue color intensities into a four-byte RGB integer for use with RGB functions and subroutines. In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value returned with RGBTOINTEGER, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

### Output

The result type is INTEGER(4). The result is the combined RGB color value.

# INTERGERTORGB

*Converts an RGB color value into its red, green, and blue components.*

### Prototype

```
INTERFACE
  SUBROUTINE INTEGER(4) (RGB,RED,GREEN,BLUE)
    INTEGER(4) RGB,RED,GREEN,BLUE
  END SUBROUTINE
END INTERFACE
```

| | |
|---|---|
| RGB | Input. `INTEGER(4)`. RGB color value whose red, green, and blue components are to be returned. |
| RED | Output. `INTEGER(4)`. Intensity of the red component of the RGB color value. |
| GREEN | Output. `INTEGER(4)`. Intensity of the green component of the RGB color value. |
| BLUE | Output. `INTEGER(4)`. Intensity of the blue component of the RGB color value. |

### Description

`INTEGERTORGB` separates the four-byte RGB color value into the three components as follows:

# Font Manipulation Functions

# GETFONTINFO

*Gets the current font characteristics.*

## Prototype

```
INTERFACE
  FUNCTION GETFONTINFO(FI)
     INTEGER(2) GETFONTINFO
     STRUCTURE /FONTINFO/
          INTEGER(4) TYPE
          INTEGER(4) ASCENT
          INTEGER(4) PIXWIDTH
          INTEGER(4) PIXHEIGHT
          INTEGER(4)4 AVGWIDTH
          CHARACTER(LEN=81) FILENAME
          CHARACTER(LEN=32) FACENAME
          LOGICAL(1) ITALIC
          LOGICAL(1) EMPHASIZED
          LOGICAL(1) UNDELINE
       END STRUCTURE
       RECORD /FONTINFO/FI
     END FUNCTION
  END INTERFACE
FI           Output. Derived type fontinfo. Set of characteristics of
             the current font.
TYPE FONTINFO
  INTEGER(4) TYPE       ! 1 = truetype, 0 = bit map
  INTEGER(4) ASCENT     ! Pixel distance from top
                        ! to baseline
  INTEGER(4) PIXWIDTH   ! Character width in pixels,
```

```
                                 ! 0=proportional
            INTEGER(4) PIXHEIGHT  ! Character height in pixels
            INTEGER(4) AVGWIDTH   ! Average character width in
                                 ! pixels
            CHARACTER(32)XFACENAME ! Font name
            LOGICAL(1) ITALIC     ! .TRUE. if current font
                                 ! formatted italic
            LOGICAL(1) EMPHASIZED ! .TRUE. if current font
                                 ! formatted bold
            LOGICAL(1) UNDERLINE  ! .TRUE. if current font
                                 ! formatted underlined
       END TYPE FONTINFO
```

### Description

You must initialize fonts with `INITIALIZEFONTS` before calling any font-related function, including `GETFONTINFO`.

### Output

The result type is `INTEGER(2)`. The result is zero if successful; otherwise, -1.

# GETGTEXTEXTENT

*Returns the width in pixels required to print a given string of text with **OUTGTEXT** using the current font.*

### Prototype

```
       INTERFACE
         FUNCTION GETGTEXTEXTENT(TEXT)
            INTEGER(2) GETGTEXTEXTENT
            CHARACTER(LEN=*) TEXT
```

```
            END FUNCTION
        END INTERFACE
```

TEXT            Input. CHARACTER(LEN=*). Text to be analyzed.

### Description

Returns the width in pixels that would be required to print a given string of text (including any trailing blanks) with OUTGTEXT using the current font. This function is useful for determining the size of text that uses proportionally spaced fonts. You must initialize fonts with INITIALIZEFONTS before calling any font-related function, including GETGTEXTEXTENT.

### Output

The result type is INTEGER(2). The result is the width of *text* in pixels if successful; otherwise, -1 (for example, if fonts have not been initialized with INITIALIZEFONTS).

# OUTGTEXT

*In graphics mode, sends a string of text to the screen, including any trailing blanks.*

### Prototype

```
        INTERFACE
          SUBROUTINE OUTGTEXT(TEXT)
            CHARACTER(LEN=*) TEXT
          END SUBROUTINE
        END INTERFACE
```

TEXT            Input. CHARACTER(LEN=*). String to be displayed.

### Description

In graphics mode, sends a string of text to the screen, including any trailing blanks. Text output begins at the current graphics position, using the current font set with SETFONT and the current color set with SETCOLORRGB or SETCOLOR. No formatting is provided. After it outputs the text, OUTGTEXT updates the current graphics position.

Before you call OUTGTEXT, you must call INITIALIZEFONTS.

Because OUTGTEXT is a graphics function, the color of text is affected by the SETCOLORRGB function, not by SETTEXTCOLORRGB.

# INITIALIZEFONTS

*Initializes Windows fonts.*

### Prototype

```
INTERFACE
  FUNCTION INITIALIZAFONTS()
    INTEGER(2) INITIALIZAFONTS
  END FUNCTION
END INTERFACE
```

### Description

All fonts in Windows become available after a call to INITIALIZEFONTS. Fonts must be initialized with INITIALIZEFONTS before any other font-related library function (such as GETFONTINFO, GETGTEXTEXTENT, SETFONT, OUTGTEXT) can be used.

For each window you open, you must call INITIALIZEFONTS before calling SETFONT. INITIALIZEFONTS needs to be executed after each new child window is opened in order for a subsequent SETFONT call to be successful.

### Output

The result type is `INTEGER(2)`. The result is the number of fonts initialized

# SETFONT

*Finds a single font that matches a specified set of characteristics.*

### Prototype

```
INTERFACE
  FUNCTION SETFONT(OPTIONS)
    INTEGER(2) SETFONT
    CHARACTER(LEN=*) OPTIONS
  END FUNCTION
END INTERFACE
```

OPTIONS      Input. `CHARACTER(LEN=*)`. String describing font characteristics

### Description

Finds a single font that matches a specified set of characteristics and makes it the current font used by the `OUTGTEXT` function. The `SETFONT` function searches the list of available fonts for a font matching the characteristics specified in `OPTIONS`. If a font matching the characteristics is found, it becomes the current font. The current font is used in all subsequent calls to the `OUTGTEXT` function. There can be only one current font.

The `OPTIONS` argument consists of letter codes, as follows, that describe the desired font. The `OPTIONS` parameter is not case sensitive or position sensitive.

`t'fontname'`    Name of the desired typeface. It can be any installed font.

`hy`          Character height, where *y* is the number of pixels.

| | |
|---|---|
| w*x* | Select character width, where *x* is the number of pixels. |
| f | Select only a fixed-space font (do not use with the p characteristic). |
| p | Select only a proportional-space font (do not use with the f characteristic). |
| v | Select only a vector-mapped font (do not use with the **r** characteristic). In Windows NT, Roman, Modern, and Script are examples of vector-mapped fonts, also called plotter fonts. True Type fonts (for example, Arial, Symbol, and Times New Roman) are not vector-mapped. |
| r | Select only a raster-mapped (bitmapped) font (do not use with the **v** characteristic). In Windows NT, Courier, Helvetica, and Palatino are examples of raster-mapped fonts, also called screen fonts. True Type fonts are not raster-mapped. |
| e | Select the bold text format. This parameter is ignored if the font does not allow the bold format. |
| u | Select the underline text format. This parameter is ignored if the font does not allow underlining. |
| i | Select the italic text format. This parameter is ignored if the font does not allow italics. |
| b | Select the font that best fits the other parameters specified. |
| n*x* | Select font number *x*, where *x* is less than or equal to the value returned by the INTIALIZEFONTS function. |

You can specify as many options as you want, except with **n***x*, which should be used alone. If you specify options that are mutually exclusive (such as the pairs f/p or r/v), the SETFONT function ignores them. There is no error detection for incompatible parameters used with **n***x*.

If the b option is specified and at least one font is initialized, SETFONT sets a font and returns 0 to indicate success.

In selecting a font, the `SETFONT` routine uses the following criteria, rated from highest precedence to lowest:

- Pixel height
- Typeface
- Pixel width
- Fixed or proportional font

You can also specify a pixel width and height for fonts. If you choose a nonexistent value for either and specify the b option, `SETFONT` chooses the closest match.

A smaller font size has precedence over a larger size. If you request Arial 12 with best fit, and only Arial 10 and Arial 14 are available, `SETFONT` selects Arial 10.

If you choose a nonexistent value for pixel height and width, the `SETFONT` function applies a magnification factor to a vector-mapped font to obtain a suitable font size. This automatic magnification does not apply if you specify the r option (raster-mapped font), or if you request a specific typeface and do not specify the b option (best-fit).

If you specify the n$x$ parameter, `SETFONT` ignores any other specified options and supplies only the font number corresponding to $x$.

If a height is given, but not a width, or vice versa, `SETFONT` computes the missing value to preserve the correct font proportions.

The font functions affect only `OUTGTEXT` and the current graphics position; no other Fortran Graphics Library output functions are affected by font usage.

For each window you open, you must call `INITIALIZEFONTS` before calling `SETFONT`. `INITIALIZEFONTS` needs to be executed after each new child window is opened in order for a subsequent `SETFONT` call to be successful.

### Output

The result type is `INTEGER(2)`. The result is the index number ($x$ as used in the n$x$ option) of the font if successful; otherwise, - 1.

# SETGTEXTROTATION

*Sets the orientation angle of the font text
output in degrees.*

### Prototype

```
INTERFACE
  SUBROUTINE SETGTEXTROTATION(DEGREES)
    INTEGER(4) DEGREES
  END SUBROUTINE
END INTERFACE
```

DEGREES          Input. INTEGER(4). Angle of orientation, in tenths of
                 degrees, of the font text output.

### Description

Sets the orientation angle of the font text output in degrees. The current
orientation is used in calls to OUTGTEXT. The orientation of the font text
output is set in tenths of degrees. Horizontal is 0°, and angles increase
counterclockwise so that 900 (90°) is straight up, 1800 (180°) is upside
down and left, 2700 (270°) is straight down, and so forth. If the user
specifies a value greater than 3600 (360°), the subroutine takes a value
equal to:

MODULO (user-specified tenths of degrees, 3600)

Although SETGTEXTROTATION accepts arguments in tenths of degrees,
only increments of one full degree differ visually from each other on the
screen.

# GETGTEXTROTATION

*Returns the current orientation of the
font text output by* OUTGTEXT.

### Prototype

```
INTERFACE
  FUNCTION GETGTEXTROTATION()
    INTEGER(4) GETGTEXTROTATION
  END FUNCTION
END INTERFACE
```

### Description

The orientation for text output with OUTGTEXT is set with
SETGTEXTROTATION.

### Output

The result type is INTEGER(4). It is the current orientation of the font text
output in tenths of degrees. Horizontal is 0°, and angles increase
counterclockwise so that 900 tenths of degrees (90°) is straight up, 1800
tenths of degrees (180°) is upside-down and left, 2700 tenths of degrees
(270°) is straight down, and so forth.

# GETTEXTCOLORRGB

*Gets the Red-Green-Blue (RGB) value*
*of the current text color.*

### Prototype

```
INTERFACE
  FUNCTION GETTEXTCOLORRGB()
     INTEGER(4) GETTEXTCOLORRGB
  END FUNCTION
END INTERFACE
```

### Description

Gets the Red-Green-Blue (RGB) value of the current text color (used with OUTTEXT, WRITE and PRINT).In each RGB color calue, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETTEXTCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

GETTEXTCOLORRGB returns the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT), set with SETTEXTCOLORRGB. The RGB color value used for graphics is set and returned with SETCOLORRGB and GETCOLORRGB. SETCOLORRGB controls the color used by the graphics function OUTGTEXT, while SETTEXTCOLORRGB controls the color used by all other text output functions. The RGB background color value for both text and graphics is set and returned with SETBKCOLORRGB and GETBKCOLORRGB.

SETTEXTCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB, and SETCOLORRGB) sets the color to a color value chosen from the entire available range. The non-RGB color functions (SETTEXTCOLOR, SETBKCOLOR, and SETCOLOR) use color indexes rather than true color values. If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). It is the RGB value of the current text color.

# SETTEXTCOLORRGB

*Sets the current text color to the*
*specified Red-Green-Blue (RGB) value.*

### Prototype

```
INTERFACE
  FUNCTION SETTEXTCOLORRGB(COLOR)
    INTEGER(4) SETTEXTCOLORRGB,COLOR
  END FUNCTION
END INTERFACE
```

COLOR            Input. INTEGER(4). RGB color value to set the text color to. Range and result depend on the system's display adapter.

### Description

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETTEXTCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, #0000FF yields full-intensity red, #00FF00 full-intensity green, #FF0000 full-intensity blue, and #FFFFFF full-intensity for all three, resulting in bright white.

SETTEXTCOLORRGB sets the current text RGB color. The default value is #00FFFFFF, which is full-intensity white. SETTEXTCOLORRGB sets the color used by OUTTEXT, WRITE, and PRINT. It does not affect the color of text output with the OUTGTEXT font routine. Use SETCOLORRGB to change the color of font output.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. SETCOLORRGB sets the RGB color value of graphics over the background color, used by the graphics functions such as ARC, FLOODFILLRGB, and OUTGTEXT.

SETTEXTCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB and SETCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETTEXTCOLOR, SETBKCOLOR, and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are limited to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

### Output

The result type is INTEGER(4). The result is the previous text RGB color value.

# QuickWin Compatible Support

# GETWINDOWCONFIG

*Gets the properties of the current window.*

**Prototype**

```
INTERFACE
  FUNCTION GETWINDOWCONFIG(WC)
    LOGICAL GETWINDOWCONFIG
    STRUCTURE /WINDOWCONFIG/
    INTEGER(2) NUMXPIXELS,NUMYPIXELS
    INTEGER(2) NUMTEXTCOLS,NUMTEXTROWS
    INTEGER(2) NUMCOLORS
    INTEGER(2) FONTSIZE
    CHARACTER(LEN=80) TITLE
    INTEGER(2) BITSPERPIXEL
    INTEGER(2) NUMVIDEOPAGES
    INTEGER(2) MODE
    INTEGER(2) ADAPTER
    INTEGER(2) MONITOR
    INTEGER(2) MEMORY
    INTEGER(2) ENVIRONMENT
    CHARACTER(LEN=32) EXTENDFONTNAME
    INTEGER(4) EXTENDFONTSIZE
    INTEGER(4) EXTENDFONTATTRIBUTES
    END STRUCTURE
    RECORD /WINDOWCONFIG/WC
  END FUNCTION
END INTERFACE
```

WC               Output. Derived type windowconfig. Contains window
                 properties.

```
TYPE WINDOWCONFIG
INTEGER(2) NUMXPIXELS    ! Number of pixels on x-axis
INTEGER(2) NUMYPIXELS    ! Number of pixels on y-axis
INTEGER(2) NUMTEXTCOLS   ! Number of text columns
                         ! available
INTEGER(2) NUMTEXTROWS   ! Number of text rows
                         ! available
INTEGER(2) NUMCOLORS     ! Number of color indexes
INTEGER(4) FONTSIZE      ! Size of default font. Set
               ! to QWIN$EXTENDFONT when using
               ! multibyte characters, in which case
               ! EXTENFONTSIZE sets the font size.
CHARACTER(LEN=80) title  ! window title
INTEGER(2) BITSPERPIXEL  ! number of bits per pixel
! The next three parameters support multibyte
! character sets (such as Japanese)
CHARACTER(LEN=32) EXTENDFONTNAME     ! any
! nonproportionally spaced font available on the
! system
INTEGER(4) EXTENDFONTSIZE ! takes same values as
               ! FONTSIZE, but used for multibyte
               ! character sets when FONTSIZE set to
               ! QWIN$EXTENDFONT
INTEGER(4) EXTENDFONTATTRIBUTES  ! font attributes
               ! such as bold and italic for
               ! multibyte character sets
END TYPE WINDOWCONFIG
```

## Description

GETWINDOWCONFIG returns information about the active child window. If
you have not set the window properties with SETWINDOWCONFIG,
GETWINDOWCONFIG returns default window values.

A typical set of values would be 1024 X pixels, 768 Y pixels, 128 text columns, 48 text rows, and a font size of 8x16 pixels. The resolution of the display and the assumed font size of 8x16 pixels generates the number of text rows and text columns. The resolution (in this case, 1024 X pixels by 768 Y pixels) is the size of the *virtual* window. To get the size of the *physical* window visible on the screen, use GETWSIZEQQ. In this case, GETWSIZEQQ returned the following values: (0,0) for the x and y position of the physical window, 25 for the height or number of rows, and 71 for the width or number of columns.

The number of colors returned depends on the video drive. The window title defaults to "Graphic1" for the default window. All of these values can be changed with SETWINDOWCONFIG.

Note that the bitsperpixel field in the windowconfig derived type is an output field only, while the other fields return output values to GETWINDOWCONFIG and accept input values from SETWINDOWCONFIG.

### Output

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE. (for example, if there is no active child window).

# SETWINDOWCONFIG

*Sets the properties of a child window.*

### Prototype

```
INTERFACE
  FUNCTION SETWINDOWCONFIG(wc)
    logical SETWINDOWCONFIG
    STRUCTURE /WINDOWCONFIG/
    INTEGER(2) NUMXPIXELS, NUMYPIXELS
    INTEGER(2) NUMTEXTCOLS, NUMTEXTROWS
    INTEGER(2) NUMCOLORS
```

```
                  INTEGER(4) FONTSIZE
                  CHARACTER(LEN=80) TITLE
                  INTEGER(2) BITSPERPIXEL
                  INTEGER(2) NUMVIDEOPAGES
                  INTEGER(2) MODE
                  INTEGER(2) ADAPTER
                  INTEGER(2) MONITOR
                  INTEGER(2) MEMORY
                  INTEGER(2) ENVIRONMENT
                  CHARACTER(LEN=32) EXTENDFONTNAME
                  INTEGER(4) EXTENDFONTSIZE
                  INTEGER(4) EXTENDFONTATTRIBUTES
                  END STRUCTURE
                  RECORD /WINDOWCONFIG/WC
              END FUNCTION
          END INTERFACE
```

WC              Input. Derived type windowconfig. Contains window
                properties.

```
 TYPE WINDOWCONFIG
  INTEGER(2) NUMXPIXELS   ! Number of pixels on x-axis
  INTEGER(2) NUMYPIXELS   ! Number of pixels on y-axis
  INTEGER(2) NUMTEXTCOLS  ! Number of text columns
                             ! available
    INTEGER(2) NUMTEXTROWS ! Number of text rows
                             ! available
    INTEGER(2) NUMCOLORS   ! Number of color indexes
    INTEGER(4) FONTSIZE    !  Size of default font. Set
                    ! to QWIN$EXTENDFONT when using
                    ! multibyte characters, in which case
                    ! EXTENDFONTSIZE sets the font size.
    CHARACTER(LEN=80) TITLE ! window title, a C string
  ! The next three parameters support multibyte
  ! character sets (such as Japanese)
```

```
   CHARACTER(LEN=32) EXTENDFONTNAME ! any
! nonproportionally spaced font available on the
! system
INTEGER(4) EXTENDFONTSIZE ! takes same values as
                ! FONTSIZE, but used for multibyte
                ! character sets when FONTSIZE set to
                ! QWIN$EXTENDFONT
INTEGER(4) EXTENDFONTATTRIBUTES  ! font attributes
                ! such as bold and italic for
                ! multibyte character sets
END TYPE WINDOWCONFIG
```

## Description

If you use `SETWINDOWCONFIG` to set the variables in `WINDOWCONFIG` to - 1, the function sets the highest resolution possible for your system, given the other fields you specify, if any. You can set the actual size of the window by specifying parameters that influence the window size: the number of x and y pixels, the number of rows and columns, and the font size. If you do not call `SETWINDOWCONFIG`, the window defaults to the best possible resolution and a font size of 8x16. The number of colors available depends on the video driver used.

If you use `SETWINDOWCONFIG`, you should specify a value for each field (-1 or your own value for the numeric fields and a C string for the title, for example, "words of text"C). Using `SETWINDOWCONFIG` with only some fields specified can result in useless values for the unspecified fields.

If you request a configuration that cannot be set, `SETWINDOWCONFIG` returns `.FALSE.` and calculates parameter values that will work and are as close as possible to the requested configuration. A second call to `SETWINDOWCONFIG` establishes the adjusted values; for example:

```
status = SETWINDOWCONFIG(WC)
```

```
if (.NOT.status) status = SETWINDOWCONFIG(WC)
```

If you specify values for all four of the size parameters, `NUMXPIXELS`, `NUMYPIXELS`, `NUMTEXTCOLS`, and `NUMTEXTROWS`, the font size is calculated by dividing these values.The default font is Courier New and the default font size is 8x16. There is no restriction on font size, except that the window must be large enough to hold it.

Under Standard Graphics, the application attempts to start in Full Screen mode with no window decoration (window decoration includes scroll bars, menu bar, title bar, and message bar) so that the maximum resolution can be fully used. Otherwise, the application starts in a window. You can use ALT+ENTER at any time to toggle between the two modes.

Note that if you are in Full Screen mode and the resolution of the window does not match the resolution of the video driver, graphics output will be slow compared to drawing in a window.

### Output

The result type is `LOGICAL(4).` The result is `.TRUE.` if successful; otherwise, `.FALSE..`

## APPENDMENUQQ

*Appends a menu item to the end of a menu and registers its callback subroutine.*

### Prototype

```
INTERFACE
  FUNCTION APPENDMENUQQ(MENUID,FLAGS,TEXT,ROUTINE)
    LOGICAL APPENDMENUQQ
    INTEGER(4) MENUID,FLAGS
    CHARACTER(LEN=*) TEXT
    EXTERNAL ROUTINE
  END FUNCTION
```

```
        END INTERFACE
```

MENUID        Input. `INTEGER(4)`. Identifies the menu to which the item is appended, starting with 1 as the leftmost menu.

FLAGS        Input. `INTEGER(4)`. Constant indicating the menu state. Flags can be combined with an inclusive **OR**. The following constants are available:

     `$MENUGRAYED`      Disables and grays out the menu item.

     `$MENUDISABLED`      Disables but does not gray out the menu item.

     `$MENUENABLED`      Enables the menu item.

     `$MENUSEPARATOR`      Draws a separator bar.

     `$MENUCHECKED`      Puts a check by the menu item.

     `$MENUUNCHECKED`      Removes the check by the menu item.

TEXT        Input. `CHARACTER(LEN=*)`. Menu item name. Must be a null-terminated C string, for example, 'WORDS OF TEXT'C.

ROUTINE        Input. `EXTERNAL`. Callback subroutine that is called if the menu item is selected. All routines take a single `LOGICAL` parameter which indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

     `WINPRINT`      Prints the program.

     `WINSAVE`      Saves the program.

     `WINEXIT`      Terminates the program.

     `WINSELTEXT`      Selects text from the current window.

     `WINSELGRAPH`      Selects graphics from the current window.

     `WINSELALL`      Selects the entire contents of the current window.

| | |
|---|---|
| WINCOPY | Copies the selected text and/or graphics from the current window to the Clipboard. |
| WINPASTE | Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ. |
| WINCLEARPASTE | Clears the paste buffer. |
| WINSIZETOFIT | Sizes output to fit window. |
| WINFULLSCREEN | Displays output in full screen. |
| WINSTATE | Toggles between pause and resume states of text output. |
| WINCASCADE | Cascades active windows. |
| WINTILE | Tiles active windows. |
| WINARRANGE | Arranges icons. |
| WINSTATUS | Enables a status bar. |
| WININDEX | Displays the index for QuickWin help. |
| WINUSING | Displays information on how to use Help. |
| WINABOUT | Displays information about the current QuickWin application. |
| NUL | No callback routine. |

### Description

You do not need to specify a menu item number, because APPENDMENUQQ always adds the new item to the bottom of the menu list. If there is no item yet for a menu, your appended item is treated as the top-level menu item (shown on the menu bar), and TEXT becomes the menu title. APPENDMENUQQ ignores the callback routine for a top-level menu item if there are any other menu items in the menu. In this case, you can set ROUTINE to NUL.

If you want to insert a menu item into a menu rather than append to the bottom of the menu list, use `INSERTMENUQQ`.

The constants available for flags can be combined with an inclusive `OR` where reasonable, for example `$MENUCHECKED .OR. $MENUENABLED`. Some combinations do not make sense, such as `$MENUENABLED` and `$MENUDISABLED`, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to `APPENDMENUQQ` as `TEXT` by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the `R` underlined, `TEXT` should be `"P&rint"`. Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY (ALT+R in the example) as an alternative to selecting the item with the mouse.

### Output

The result type is `LOGICAL:` `.TRUE.` if successful; otherwise, `.FALSE.`.

# INSERTMENUQQ

*Inserts a menu item into a QuickWin menu and registers its callback routine.*

### Prototype

```
INTERFACE
 FUNCTION
  INSERTMENUQQ(MENUID,ITEMID,FLGS,TEXT,ROUTINE)
    LOGICAL INSERTMENUQQ
    INTEGER(4) MENUID,ITEMID,FLAGS
    CHARACTER(LEN=*) TEXT
    EXTERNAL ROUTINE
 END FUNCTION
END INTERFACE
```

MENUID    Input. `INTEGER(4)`. Identifies the menu in which the item is inserted, starting with 1 as the leftmost menu.

ITEMID    Input. `INTEGER(4)`. Identifies the position in the menu where the item is inserted, starting with 0 as the top menu item.

FLAGS     Input. `INTEGER(4)`. Constant indicating the menu state. Flags can be combined with an inclusive OR (see Results section below). The following constants are available:

| | |
|---|---|
| `$MENUGRAYED` | Disables and grays out the menu item. |
| `$MENUDISABLED` | Disables but does not gray out the menu item. |
| `$MENUENABLED` | Enables the menu item. |
| `$MENUSEPARATOR` | Draws a separator bar. |
| `$MENUCHECKED` | Puts a check by the menu item. |
| `$MENUUNCHECKED` | Removes the check by the menu item. |

TEXT      Input. `CHARACTER(LEN=*)`. Menu item name. Must be a null-terminated C string, for example, words of text'C.

ROUTINE   Input. `EXERNAL`. Callback subroutine that is called if the menu item is selected. All routines must take a single `LOGICAL` parameter which indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

| | |
|---|---|
| `WINPRINT` | Prints the program. |
| `WINSAVE` | Saves the program. |
| `WINEXIT` | Terminates the program. |
| `WINSELTEXT` | Selects text from the current window. |
| `WINSELGRAPH` | Selects graphics from the current window. |

| | |
|---|---|
| WINSELALL | Selects the entire contents of the current window. |
| WINCOPY | Copies the selected text and/or graphics from current window to the Clipboard. |
| WINPASTE | Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ. |
| WINCLEARPASTE | Clears the paste buffer. |
| WINSIZETOFIT | Sizes output to fit window. |
| WINFULLSCREEN | Displays output in full screen. |
| WINSTATE | Toggles between pause and resume states of text output. |
| WINCASCADE | Cascades active windows. |
| WINTILE | Tiles active windows. |
| WINARRANGE | Arranges icons. |
| WINSTATUS | Enables a status bar. |
| WININDEX | Displays the index for QuickWin help. |
| WINUSING | Displays information on how to use Help. |
| WINABOUT | Displays information about the current QuickWin application. |
| NUL | No callback routine. |

### Description

Menus and menu items must be defined in order from left to right and top to bottom. For example, INSERTMENUQQ fails if you try to insert menu item 7 when 5 and 6 are not defined yet. For a top-level menu item, the callback routine is ignored if there are subitems under it.

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to INSERTMENUQQ as *text* by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, *text* should be "P&rint". Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY (ALT+R in the example) as an alternative to selecting the item with the mouse.

### Output

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

# DELETEMENUQQ

*Deletes a menu item from a QuickWin menu.*

### Prototype

```
INTERFACE
  FUNCTION DELETEMENUQQ(MENUID,ITEMID)
    LOGICAL DELETEMENUQQ
    INTEGER(4) MENUID,ITEMID
  END FUNCTION
END INTERFACE
```

MENUID          Input. INTEGER(4). Identifies the menu that contains the menu item to be deleted, starting with 1 as the leftmost menu.

| | |
|---|---|
| ITEMID | Input. `INTEGER(4)`. Identifies the menu item to be deleted, starting with 0 as the top menu item. |

### Description

Deletes a menu item from a QuickWin menu.

### Output

The result type is `LOGICAL(4)`. The result is `.TRUE.` if successful; otherwise, `.FALSE.`.

# MODIFYMENUFLAGSQQ

*Modifies a menu item's state.*

### Prototype

```
INTERFACE
 FUNCTION MODIFYMENUFLAGSQQ(MENUID,ITEMID,FLAGS)
   LOGICAL MODIFYMENUFLAGSQQ
   INTEGER(4) MENUID,ITEMID,FLAGS
 END FUNCTION
END INTERFACE
```

| | |
|---|---|
| MENUID | Input. `INTEGER(4)`. Identifies the menu containing the item whose state is to be modified, starting with 1 as the leftmost menu. |
| ITEMID | Input. `INTEGER(4)`. Identifies the menu item whose state is to be modified, starting with 0 as the top item. |
| FLAGS | Input. `INTEGER(4)`. Constant indicating the menu state. Flags can be combined with an inclusive `OR`. The following constants are available: |
| | `$MENUGRAYED`  Disables and grays out the menu item. |

| | |
|---|---|
| `$MENUDISABLED` | Disables but does not gray out the menu item. |
| `$MENUENABLED` | Enables the menu item. |
| `$MENUSEPARATOR` | Draws a separator bar. |
| `$MENUCHECKED` | Puts a check by the menu item. |
| `$MENUUNCHECKED` | Removes the check by the menu item. |

### Description

The constants available for flags can be combined with an inclusive `OR` where reasonable, for example `$MENUCHECKED .OR. $MENUENABLED`. Some combinations do not make sense, such as `$MENUENABLED` and `$MENUDISABLED`, and lead to undefined behavior.

### Output

The result type is `LOGICAL(4)`. The result is `.TRUE.` if successful; otherwise, `.FALSE.`.

# MODIFYMENUSTRINGQQ

*Changes a menu item's text string.*

### Prototype

```
INTERFACE
 FUNCTION MODIFYMENUSTRINGQQ(MENUID,ITEMID,TEXT)
    LOGICAL MODIFYMENUSTRINGQQ
    INTEGER(4) MENUID,ITEMID
    CHARACTER(LEN=*) TEXT
 END FUNCTION
END INTERFACE
```

MENUID          Input. `INTEGER(4)`. Identifies the menu containing the item whose text string is to be changed, starting with 1 as the leftmost item.

ITEMID          Input. `INTEGER(4)`. Identifies the menu item whose text string is to be changed, starting with 0 as the top menu item.

TEXT            Input. `CHARACTER(LEN=*)`. Menu item name. Must be a null-terminated C string. For example, words of text'C.

### Description

You can add access keys in your text strings by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, use `"P&rint"`C as `TEXT`.

### Output

The result type is `LOGICAL(4)`. The result is `.TRUE.` if successful; otherwise, `.FALSE.`.

# MODIFYMENUROUTINEQQ

*Changes a menu item's callback routine.*

### Prototype

```
INTERFACE
 FUNCTION MODIFYMENUROUTINEQQ(MENUID,ITENID,ROUTINE)
   LOGICAL MODIFYMENUROUTINEQQ
   INTEGR(4) MENUID,ITEMID
   EXTERNAL ROUTINE
 END FUNCTION
END INTERFACE
```

MENUID      Input. `INTEGER(4)`. Identifies the menu that contains the item whose callback routine is be changed, starting with 1 as the leftmost menu.

ITEMID      Input. `INTEGER(4)`. Identifies the menu item whose callback routine is to be changed, starting with 0 as the top item.

ROUTINE      Input. `EXTERNAL`. Callback subroutine called if the menu item is selected. All routines must take a single `LOGICAL` parameter that indicates whether the menu item is checked or not. The following predefined routines are available for assigning to menus:

| | |
|---|---|
| `WINPRINT` | Prints the program. |
| `WINSAVE` | Saves the program. |
| `WINEXIT` | Terminates the program. |
| `WINSELTEXT` | Selects text from the current window. |
| `WINSELGRAPH` | Selects graphics from the current window. |
| `WINSELALL` | Selects the entire contents of the current window. |
| `WINCOPY` | Copies the selected text and/or graphics from the current window to the Clipboard. |
| `WINPASTE` | Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a `READ`. |
| `WINCLEARPASTE` | Clears the paste buffer. |
| `WINSIZETOFIT` | Sizes output to fit window. |
| `WINFULLSCREEN` | Displays output in full screen. |
| `WINSTATE` | Toggles between pause and resume states of text output. |

| | |
|---|---|
| WINCASCADE | Cascades active windows. |
| WINTILE | Tiles active windows. |
| WINARRANGE | Arranges icons. |
| WINSTATUS | Enables a status bar. |
| WININDEX | Displays the index for QuickWin Help. |
| WINUSING | Displays information on Help. |
| WINABOUT | Displays information about the current QuickWin application. |
| NUL | No callback routine. |

### Description

This function changes a menu item's callback routine.

### Output

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

## SETWINDOWMENUQQ

*Sets a top-level menu as the menu to which a list of current child window names is appended.*

### Prototype

```
INTERFACE
  FUNCTION SETWINDOWMENUQQ(MENUID)
    LOGICAL SETWINDOWMENUQQ
    INTEGER(4) MENUID
  END FUNCTION
END INTERFACE
```

MENUID                    Input. INTEGER(4). Identifies the menu to hold the child window names, starting with 1 as the leftmost menu.

### Description

The list of current child window names can appear in only one menu at a time. If the list of windows is currently in a menu, it is removed from that menu. By default, the list of child windows appears at the end of the Window menu.

### Output

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

# SETACTIVEQQ

*Makes a child window active, but does not give it focus.*

### Prototype

```
INTERFACE
   FUNCTION SETACTIVEQQ(UNIT)
      INTEGER(4) SETACTIVEQQ, UNIT
   END FUNCTION
END INTERFACE
```

UNIT                      Input. INTEGER(4). Unit number of the child window to be made active.

### Description

When a window is made active, it receives graphics output (from ARC, LINETO and OUTGTEXT, for example) but is not brought to the foreground and does not have the focus. If a window needs to be brought to the foreground, it must be given the focus. A window is given focus with

FOCUSQQ, by clicking it with the mouse, or by performing I/O other than graphics on it, unless the window was opened with IOFOCUS='.FALSE.'. By default, IOFOCUS='.TRUE.', except for child windows opened as unit '*'.

The window that has the focus is always on top, and all other windows have their title bars grayed out. A window can have the focus and yet not be active and not have graphics output directed to it. Graphical output is independent of focus.

If IOFOCUS='.TRUE.', the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

### Output

The result type is INTEGER(4). The result is 1 if successful; otherwise, 0.

# GETACTIVEQQ

*Returns the unit number of the currently active child window.*

### Prototype

```
INTERFACE
  FUNCTION GETACTIVEQQ()
    INTEGER(4) GETACTIVEQQ
  END FUNCTION
END INTERFACE
```

### Description

This function returns the unit number of the currently active child window.

**Output**

The result type is INTEGER(4). The result is the unit number of the currently active window. Returns the parameter QWIN$NOACTIVEWINDOW if no child window is active.

# FOCUSQQ

*Sets focus to the window with the specified unit number.*

**Prototype**

```
INTERFACE
  FUNCTION FOCUSQQ(IUNIT)
    INTEGER(4) FOCUSQQ,IUNIT
  END FUNCTION
END INTERFACE
```

IUNIT          Input. INTEGER(4). Unit number of the window to which the focus is set. Unit numbers 0, 5, and 6 refer to the default startup window.

**Description**

Units 0, 5, and 6 refer to the default window only if the program does not specifically open them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

Unlike SETACTIVEQQ, FOCUSQQ brings the specified unit to the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling SETACTIVEQQ.

A window has focus when it is given the focus by FOCUSQQ, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with

`IOFOCUS=.FALSE.`. The `IOFOCUS` specifier determines whether a window receives focus when on I/O statement is executed on that unit. For example:

`OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)`

By default `IOFOCUS=.TRUE.`, except for child windows opened with as unit *. If `IOFOCUS=.TRUE.`, the child window receives focus prior to each `READ`, `WRITE`, `PRINT`, or `OUTTEXT`. Calls to graphics functions (such as `OUTGTEXT` and `ARC`) do not cause the focus to shift.

### Output

The result type is `INTEGER(4)`. The result is zero if successful; otherwise, nonzero.

# INQFOCUSQQ

*Determines which window has the focus.*

### Prototype

```
INTERFACE
   FUNCTION INQFOCUSQQ(IUNIT)
      INTEGER(4) INQFOCUSQQ,IUNIT
   END FUNCTION
END INTERFACE
```

IUNIT           Output. `INTEGER(4)`. Unit number of the window that has the I/O focus.

### Description

Unit numbers 0, 5, and 6 refer to the default window only if the program has not specifically opened them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

The window with focus is always in the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling SETACTIVEQQ.

A window has focus when it is given the focus by FOCUSQQ, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. The IOFOCUS specifier determines whether a window receives focus when on I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

By default IOFOCUS=.TRUE., except for child windows opened with as unit *. If IOFOCUS=.TRUE., the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

### Output

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero. The function fails if the window with the focus is associated with a closed unit.

# GETHWNDQQ

*Converts a window unit number into a Windows handle.*

### Prototype

```
INTERFACE
  FUNCTION GETHWNDQQ(IUNIT)
    INTEGER(4) GETHWNDQQ, IUNIT
  END FUNCTION
END INTERFACE
```

IUNIT                Input. `INTEGER(4)`. Window unit number. If `IUNIT` is set to `QWIN$FRAMEWINDOW`, the handle of the frame window is returned.

### Description

This function converts a window unit number into a Windows handle.

### Output

The result type is `INTEGER(4)`. The result is a true Windows handle to the window. It returns -1 if `IUNIT` is not open.

# GETUNITQQ

*Returns the unit number corresponding to the specified Windows handle.*

### Prototype

```
INTERFACE
  FUNCTION GETUNITQQ(IHANDLE)
    INTEGER(4) GETUNITQQ,IHANDLE
  END FUNCTION
END INTERFACE
```

IHANDLE              Input. `INTEGER(4)`. The Windows handle to the window; this is a unique ID.

### Description

Returns the unit number corresponding to the specified Windows handle. This routine is the inverse of `GETHWNDQQ`.

### Output

The result type is `INTEGER(4)`. The result is the unit number corresponding to the specified Windows handle. It returns -1 if `IHANDLE` does not exist.

# ABOUTBOXQQ

*Specifies the message box information
that appears when About command
from a Help menu is selected.*

### Prototype

```
INTERFACE
    FUNCTION ABOUTBOXQQ(STR)
        INTEGER(4) ABOUTBOXQQ
        CHARACTER(LEN=*) STR
    END FUNCTION
END INTERFACE
```

STR             Input; output. CHARACTER(LEN=*). Null-terminated C
                string.

### Description

Specifies the information displayed in the message box that appears when
the About command from a QuickWin application's Help menu is selected.
If your program does not call ABOUTBOXQQ, the QuickWin run-time library
supplies a default string.

### Output

The value of the result is INTEGER(4). It is zero if successful; otherwise,
nonzero.

# CLICKMENUQQ

*Simulates the effect of clicking or*
*selecting a menu command.*

### Prototype

```
INTERFACE
  FUNCTION CLICKMENUQQ(ITEM)
    INTEGER(4) CLICKMENUQQ,ITEM
  END FUNCTION
END INTERFACE
```

ITEM  Input. INTEGER(4). Constant that represents the command selected from the Window menu. Must be one of the following symbolic constants:

| | |
|---|---|
| QWIN$STATUS | Status command |
| QWIN$TILE | Tile command |
| QWIN$CASCADE | Cascade command |
| QWIN$ARRANGE | Arrange Icons command |

### Description

Simulates the effect of clicking or selecting a menu command. The QuickWin application responds as though the user had clicked or selected the command.

### Output

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

# SETWSIZEQQ

*Sets the size and position of a window.*

## Prototype

```
INTERFACE
  FUNCTION SETWSIZEQQ(IUNIT,WINFO)
    STRUCTURE /QWINFO/
        INTEGER(2) TYPE,X,Y,H,W
    END STRUCTURE
    INTEGER(4) SETWSIZEQQ, IUNIT
    RECORD /QWINFO/ WINFO
  END FUNCTION
END INTERFACE
```

IUNIT            Input. INTEGER(4). Specifies the window unit. Unit numbers 0, 5, and 6 refer to the default startup window only if the program does not explicitly open them with the OPEN statement. To set the size of the frame window (as opposed to a child window), set IUNIT to the symbolic constant QWIN$FRAMEWINDOW.

WINFO            Input. Derived type qwinfo. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame).

```
 TYPE QWINFO
    INTEGER(2) TYPE   ! request type
    INTEGER(2) X      ! x coordinate for upper left
    INTEGER(2) Y      ! y coordinate for upper left
    INTEGER(2) H      ! window height
    INTEGER(2) W      ! window width
 END TYPE QWINFO
```

This function's behavior depends on the value of qwinfo.type, which can be any of the following:

| | |
|---|---|
| QWIN$MIN | Minimizes the window. |
| QWIN$MAX | Maximizes the window. |
| QWIN$RESTORE | Restores the minimized window to its previous size. |
| QWIN$SET | Sets the window's position and size according to the other values in qwinfo. |

### Description

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width specified for a frame window reflects the actual size in pixels of the frame window *including* any borders, menus, and status bar at the bottom.

### Output

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

# GETWSIZEQQ

*Gets the size and position of a window.*

### Prototype

```
INTERFACE
  FUNCTION GETWSIZEQQ(IUNIT,IREQ,WINFO)
    structure /QWINFO/
        INTEGER(2) TYPE,X,Y,H,W
      END STRUCTURE
      INTEGER(4) GETWSIZEQQ,IUNIT
      INTEGER(4) IREQ
```

```
                    RECORD /QWINFO/ WINFO
                END FUNCTION
             END INTERFACE
```

IUNIT  
Input. `INTEGER(4)`. Specifies the window unit. Unit numbers 0, 5 and 6 refer to the default startup window only if you have not explicitly opened them with the `OPEN` statement. To access information about the frame window (as opposed to a child window), set *unit* to the symbolic constant `QWIN$FRAMEWINDOW`.

IREQ  
Input. `INTEGER(4)`. Specifies what information is obtained.

    `QWIN$SIZEMAX`    Gets information about the maximum window size.

    `QWIN$SIZECURR`    Gets information about the current window size.

WINFO  
Output. Derived type qwinfo. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame).

```
TYPE QWINFO
  INTEGER(2) TYPE  ! request type (controls
                   ! SETWSIZEQQ)
  INTEGER(2) X     ! x coordinate for upper left
  INTEGER(2) Y     ! y coordinate for upper left
  INTEGER(2) H     ! window height
  INTEGER(2) W     ! window width
END TYPE QWINFO
```

## Description

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width returned for a frame window reflects the size in pixels of the client area *excluding* any borders, menus, and status bar at the bottom of the frame window. You should adjust the values used in SETWSIZEQQ to take this into account.

The client area is the area actually available to place child windows.

### Output

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

# MESSAGEBOXQQ

*Displays a message box in a QuickWin window.*

### Prototype

```
INTERFACE
  FUNCTION MESSAGEBOXQQ(MSG,CAPTION,MTYPE)
    CHARACTER(LEN=*) MSG,CAPTION
    INTEGER(4)  MESSAGEBOXQQ,MTYPE
  END FUNCTION
END INTERFACE
```

| | |
|---|---|
| MSG | Input. CHARACTER(LEN=*). Null-terminated C string. Message the box displays. |
| CAPTION | Input. CHARACTER(LEN=*). Null-terminated C string. Caption that appears in the title bar. |
| MTYPE | Input. INTEGER(4). Symbolic constant that determines the objects (buttons and icons) and attributes of the message box. You can combine several constants using an inclusive OR (IOR or OR). The symbolic constants and their associated objects or attributes are: |

| | |
|---|---|
| `MB$ABORTRETRYIGNORE` | The Abort, Retry, and Ignore buttons. |
| `MB$DEFBUTTON1` | The first button is the default. |
| `MB$DEFBUTTON2` | The second button is the default. |
| `MB$DEFBUTTON3` | The third button is the default. |
| `MB$ICONASTERISK` | Lowercase *i* in blue circle icon. |
| `MB$ICONEXCLAMATION` | The exclamation-mark icon. |
| `MB$ICONHAND` | The stop-sign icon. |
| `MB$ICONINFORMATION` | Lowercase *i* in blue circle icon. |
| `MB$ICONQUESTION` | The question-mark icon. |
| `MB$ICONSTOP` | The stop-sign icon. |
| `MB$OK` | The OK button. |
| `MB$OKCANCEL` | The OK and Cancel buttons. |
| `MB$RETRYCANCEL` | The Retry and Cancel buttons. |
| `MB$SYSTEMMODAL` | Box is system-modal: all applications are suspended until the user responds. |
| `MB$YESNO` | The Yes and No buttons. |
| `MB$YESNOCANCEL` | The Yes, No, and Cancel buttons. |

## Description

This function displays a message box in a QuickWin window.

**Output**

The result type is INTEGER(4). The result is zero if memory is not sufficient for displaying the message box. Otherwise, the result is one of the following values, indicating the user's response to the message box:

MB$IDABORT     The Abort button was pressed.

MB$IDCANCEL    The Cancel button was pressed.

MB$IDIGNORE    The Ignore button was pressed.

MB$IDNO        The No button was pressed.

MB$IDOK        The OK button was pressed.

MB$IDRETRY     The Retry button was pressed.

MB$IDYES       The Yes button was pressed.

# GETEXITQQ

*Gets the setting for a QuickWin application's exit behavior.*

**Prototype**

```
INTERFACE
  FUNCTION GETEXITQQ()
     INTEGER(4) GETEXITQQ
  END FUNCTION
END INTERFACE
```

**Description**

This function gets the setting for a QuickWin application's exit behavior.

**Output**

The result type is INTEGER(4). The result is exit mode with one of the following constants:

QWIN$EXITPROMPT      Displays a message box that reads "Program exited with exit status *n*. Exit Window?", where *n* is the exit status from the program. If the user chooses Yes, the application closes the window and terminates. If the user chooses No, the dialog box disappears and the user can manipulate the window as usual. The user must then close the window manually.

QWIN$EXITNOPERSIST

     Terminates the application without displaying a message box.

QWIN$EXITPERSIST      Leaves the application open without displaying a message box.

The default for both QuickWin and Console Graphics applications is QWIN$EXITPROMPT.

# SETEXITQQ

*Sets a QuickWin application's exit behavior.*

### Prototype

```
INTERFACE
  FUNCTION SETEXITQQ(EXITMODE)
     INTEGER(4) SETEXITQQ,EXITMODE
  END FUNCTION
END INTERFACE
```

EXITMODE      Input. INTEGER(4). Determines the program exit behavior. The following exit parameters:

     QWIN$EXITPROMPT

         Displays the following message box: "Program exited with exit status X. Exit

Window?" where X is the exit status from the program. If Yes is entered, the application closes the window and terminates. If No is entered, the dialog box disappears and you can manipulate the windows as usual. You must then close the window manually.

QWIN$EXITNOPERSIST
Terminates the application without displaying a message box.

QWIN$EXITPERSIST
Leaves the application open without displaying a message box.

### Description

This function sets a QuickWin application's exit behavior.

### Output

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value. The default for both QuickWin and Standard Graphics applications is QWIN$EXITPROMPT.

# INCHARQQ

*Reads a single character input from the keyboard and returns the ASCII value of that character without any buffering.*

### Prototype

```
INTERFACE
  FUNCTION INCHARQQ()
    INTEGER*2 INCHARQQ
  END FUNCTION
```

```
END INTERFACE
```

### Description

Reads a single character input from the keyboard and returns the ASCII value of that character without any buffering. The keystroke is read from the child window that currently has the focus. You must call INCHARQQ before the keystroke is made (INCHARQQ does not read the keyboard buffer). This function does not echo its input. For function keys, INCHARQQ returns 0xE0 as the upper 8 bits, and the ASCII code as the lower 8 bits.

### Output

The result type is INTEGER(2). The result is the ASCII key code.

# SETMESSAGEQQ

*Changes QuickWin status messages, state messages, and dialog box messages.*

### Prototype

```
INTERFACE
  SUBROUTINE SETMESSAGEQQ(MSG,ID)
    CHARACTER(LEN=*) MSG
    INTEGER(4) ID
  END SUBROUTINE
END INTERFACE
```

MSG            Input. CHARACTER(LEN=*). Message to be displayed. Must be a regular Fortran string, not a C string. Can include multibyte characters.

ID          Input. `INTEGER(4)`. Identifier of the message to be changed. The following table shows the messages that can be changed and their identifiers:

| ID | Message |
| --- | --- |
| `QWIN$MSG_TERM` | "Program terminated with exit code" |
| `QWIN$MSG_EXITQ` | "\nExit Window?" |
| `QWIN$MSG_FINISHED` | "Finished" |
| `QWIN$MSG_PAUSED` | "Paused" |
| `QWIN$MSG_RUNNING` | "Running" |
| `QWIN$MSG_FILEOPENDLG` | "Text Files(*.txt), *.txt; Data Files(*.dat), *.dat; All Files(*.*), *.*;" |
| `QWIN$MSG_BMPSAVEDLG` | "Bitmap Files(*.bmp), *.bmp; All Files(*.*), *.*;" |
| `QWIN$MSG_INPUTPEND` | "Input pending in" |
| `QWIN$MSG_PASTEINPUTPEND` | "Paste input pending" |
| `QWIN$MSG_MOUSEINPUTPEND` | "Mouse input pending in" |
| `QWIN$MSG_SELECTTEXT` | "Select Text in" |
| `QWIN$MSG_SELECTGRAPHICS` | "Select Graphics in" |
| `QWIN$MSG_PRINTABORT` | "Error! Printing Aborted." |
| `QWIN$MSG_PRINTLOAD` | "Error loading printer driver" |
| `QWIN$MSG_PRINTNODEFAULT` | "No Default Printer." |
| `QWIN$MSG_PRINTDRIVER` | "No Printer Driver." |
| `QWIN$MSG_PRINTINGERROR` | "Print: Printing Error." |
| `QWIN$MSG_PRINTING` | "Printing" |
| `QWIN$MSG_PRINTCANCEL` | "Cancel" |
| `QWIN$MSG_PRINTINPROGRESS` | "Printing in progress..." |
| `QWIN$MSG_HELPNOTAVAIL` | "Help Not Available for Menu Item" |
| `QWIN$MSG_TITLETEXT` | "Graphic" |

### Description

You can change any string produced by QuickWin by calling SETMESSAGEQQ with the appropriate ID. This includes status messages displayed at the bottom of a QuickWin application, state messages (such as "Paused"), and dialog box messages. These messages can include multibyte characters.

# WAITONMOUSEEVENT

*Waits for the specified mouse input from the user.*

### Prototype

```
INTERFACE
 FUNCTION WAITONMOUSEEVENT(MouseEvents,KeyState,X,Y)
   INTEGER WAITONMOUSEEVENT,MouseEvents,KeyState,X,Y
 END FUNCTION
END INTERFACE
```

MOUSEEVENTS          Input. INTEGER(4). One or more mouse events that must occur before the function returns. Symbolic constants for the possible mouse events are:

MOUSE$LBUTTONDOWN
        Left mouse button down

MOUSE$LBUTTONUP
        Left mouse button up

MOUSE$LBUTTONDBLCLK
        Left mouse button double-click

MOUSE$RBUTTONDOWN
        Right mouse button down

MOUSE$RBUTTONUP
        Right mouse button up

MOUSE$RBUTTONDBLCLK
> Right mouse button double-click

MOUSE$MOVE
> Mouse moved

KEYSTATE    Output. INTEGER(4). Bitwise inclusive OR of the state of the mouse duing the event. The value returned in KEYSTATE can be any or all of the following symbolic constants:

MOUSE$KS_LBUTTON
> Left mouse button down during event

MOUSE$KS_RBUTTON
> Right mouse button down during event

MOUSE$KS_SHIFT
> SHIFT key held down during event

MOUSE$KS_CONTROL
> CONTROL key held down during event

X    Output. INTEGER(4). X position of the mouse when the event occurred.

Y    Output. INTEGER(4). Y position of the mouse when the event occurred.

### Description

WAITONMOUSEEVENT does not return until the specified mouse input is received from the user. While waiting for a mouse event to occur, the status bar changes to read "Mouse input pending in XXX" where XXX is the name of the window. When a mouse event occurs, the status bar returns to its previous value.

A mouse event must happen in the window that had focus when WAITONMOUSEEVENT was initially called. Mouse events in other windows will not end the wait. Mouse events in other windows cause callbacks to be called for the other windows, if callbacks were previously registered for those windows.

For every `BUTTONDOWN` or `BUTTONDBLCLK` event there is an associated `BUTTONUP` event. When the user double clicks, four events happen: `BUTTONDOWN` and `BUTTONUP` for the first click, and `BUTTONDBLCLK` and `BUTTONUP` for the second click. The difference between getting `BUTTONDBLCLK` and `BUTTONDOWN` for the second click depends on whether the second click occurs in the double click interval, set in the system's `CONTROL PANEL/MOUSE`.

### Output

The result type is `INTEGER(4)`. The result is the symbolic constant associated with the mouse event that occurred if successful. If the function fails, it returns the constant `MOUSE$BADEVENT`, meaning the event specified is not supported.

# REGISTERMOUSEEVENT

*Registers the application-supplied callback routine to be called when a specified mouse event occurs in a specified window.*

### Prototype

```
INTERFACE
 FUNCTION
REGISTERMOUSEEVENT(UNIT,MouseEvents,CallBackRoutine)
   INTEGER REGISTERMOUSEEVENT,UNIT
   INTEGER MouseEvents
   EXTERNAL CallBackRoutine
 END FUNCTION
END INTERFACE
```

UNIT                   Input. `INTEGER(4)`. Unit number of the window whose callback routine on mouse events is to be registered.

MOUSEEVENTS        Input. `INTEGER(4)`. One or more mouse events to be handled by the callback routine to be registered. Symbolic constants for the possible mouse events are:

`MOUSE$LBUTTONDOWN`

Left mouse button down

`MOUSE$LBUTTONUP`

Left mouse button up

`MOUSE$LBUTTONDBLCLK`

Left mouse button double-click

`MOUSE$RBUTTONDOWN`

Right mouse button down

`MOUSE$RBUTTONUP`

Right mouse button up

`MOUSE$RBUTTONDBLCLK`

Right mouse button double-click

`MOUSE$MOVE`    Mouse moved

`CALLBACKROUTINE`

Input. `EXTERNAL`. Routine to be called on specified mouse event in the specified window.

## Description

Registers the application-supplied callback routine to be called when a specified mouse event occurs in a specified window. For every `BUTTONDOWN` or `BUTTONDBLCLK` event there is an associated `BUTTONUP` event. When the user double clicks, four events happen: `BUTTONDOWN` and `BUTTONUP` for the first click, and `BUTTONDBLCLK` and `BUTTONUP` for the second click. The difference between getting `BUTTONDBLCLK` and `BUTTONDOWN` for the second click depends on whether the second click occurs in the double click interval, set in the system's `CONTROL PANEL/MOUSE`.

### Output

The result type is `INTEGER(4)`. The result is zero or a positive integer if successful; otherwise, a negative integer that can be one of the following:

`MOUSE$BADUNIT` The unit specified is not open, or is not associated with a QuickWin window.

`MOUSE$BADEVENT`

The event specified is not supported.

# UNREGISTERMOUSEEVENT

*Removes the callback routine registered*
*for a specified window by an earlier call*
*to* REGISTERMOUSEEVENT.

### Prototype

```
INTERFACE
 FUNCTION UNREGISTERMOUSEEVENT(UNIT,MOUSEEVENTS)
  INTEGER UNREGISTERMOUSEEVENT,UNIT,MOUSEEVENTS
 END FUNCTION
END INTERFACE
```

UNIT          Input. `INTEGER(4)`. Unit number of the window whose callback routine on mouse events is to be unregistered.

MOUSEEVENTS   Input. `INTEGER(4)`. One or more mouse events handled by the callback routine to be unregistered. Symbolic constants for the possible mouse events are:

`MOUSE$LBUTTONDOWN`

Left mouse button down

`MOUSE$LBUTTONUP`

Left mouse button up

`MOUSE$LBUTTONDBLCLK`

Left mouse button double-click

MOUSE$RBUTTONDOWN
> Right mouse button down

MOUSE$RBUTTONUP
> Right mouse button up

MOUSE$RBUTTONDBLCLK
> Right mouse button double-click

MOUSE$MOVE    Mouse moved

### Description

Once you call UNREGISTERMOUSEEVENT, QuickWin no longer calls the callback routine specified earlier for the window when mouse events occur. Calling UNREGISTERMOUSEEVENT when no callback routine is registered for the window has no effect.

### Output

The result type is INTEGER(4). The result is zero or a positive integer if successful; otherwise, a negative integer which can be one of the following:

MOUSE$BADUNIT The unit specified is not open, or is not associated with a QuickWin window.

MOUSE$BADEVENT
> The event specified is not supported.

# QuickWin Default Menu Support

These are predefined callback functions, which you can register as callback routines or call directly from your program.

# WINPRINT

*Prints the program*

### Prototype

```
        INTERFACE
         SUBROUTINE WINPRINT()
!MS$ ATTRIBUTES stdcall, alias:'_WINPRINT@0' ::
WINPRINT
            END SUBROUTINE
        END INTERFACE
```

### Description

This callback routine calls "Print..." dialog to print or save as bitmap file contents of current active (focused) child window.

# WINSAVE

*Saves the program*

### Prototype

```
        INTERFACE
         SUBROUTINE WINSAVE()
!MS$ ATTRIBUTES stdcall, alias:'_WINSAVE@0' ::
WINSAVE
            END SUBROUTINE
        END INTERFACE
```

### Description

This callback routine calls "Save..." dialog to print or save as bitmap file contents of current active (focused) child window.

# WINEXIT

*Terminates the program*

### Prototype

```
      INTERFACE
       SUBROUTINE WINEXIT()
!MS$ ATTRIBUTES stdcall, alias:'_WINEXIT@0' ::
WINEXIT
        END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine calls "Exit..." dialog to terminate the current active (focused) child window.

# WINCOPY

*Copies the selected text and/or graphics from current window to the Clipboard.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINCOPY()
!MS$ ATTRIBUTES stdcall, alias:'_WINCOPY@0' ::
WINCOPY
        END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine copies the selected text and/or graphics from current window to the Clipboard.

# WINPASTE

*Pastes clipboard contents (text only) to the current text window of the active window during a* READ.

### Prototype

```
      INTERFACE
       SUBROUTINE WINPASTE()
!MS$ ATTRIBUTES stdcall, alias:'_WINPASTE@0' ::  WIN-
PASTE
          END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine pastes clipboard contents (text only) to the current text window of the active window during a READ.

# WINSIZETOFIT

*Sizes output to fit window.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINSIZETOFIT()
```

```
!MS$ ATTRIBUTES stdcall, alias:'_WINSIZETOFIT@0' ::
WINSIZETOFIT
        END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine sizes output to fit window.

## WINFULLSCREEN

*Displays output in full screen.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINFULLSCREEN()
!MS$ ATTRIBUTES stdcall, alias:'_WINFULLSCREEN@0' ::
WINFULLSCREEN
        END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine displays output in full screen.

## WINSTATE

*Toggles between pause and resume states of text output.*

### Prototype

```
      INTERFACE
```

```
        SUBROUTINE WINSTATE()
!MS$ ATTRIBUTES stdcall, alias:'_WINSTATE@0' ::
WINSTATE
        END SUBROUTINE
    END INTERFACE
```

### Description

This callback routine toggles between pause and resume states of text output.

# WINCASCADE

*Cascades active windows.*

### Prototype

```
    INTERFACE
     SUBROUTINE WINCASCADE()
!MS$ ATTRIBUTES stdcall, alias:'_WINCASCADE@0' ::
WINCASCADE
        END SUBROUTINE
    END INTERFACE
```

### Description

This callback routine cascades active windows.

# WINTILE

*Tiles active windows.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINTILE()
!MS$ ATTRIBUTES stdcall, alias:'_WINTILE@0' ::
WINTILE
        END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine tiles active windows.

# WINARRANGE

*Arranges icons.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINARRANGE()
!MS$ ATTRIBUTES stdcall, alias:'_WINARRANGE@0' ::
WINARRANGE
        END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine arranges icons.

# WININPUT

### Prototype

```
      INTERFACE
       SUBROUTINE WININPUT()
!MS$ ATTRIBUTES stdcall, alias:'_WININPUT@0' ::
WININPUT
          END SUBROUTINE
      END INTERFACE
```

# WINCLEARPASTE

*Clears the paste buffer.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINCLEARPASTE()
!MS$ ATTRIBUTES stdcall, alias:'_WINCLEARPASTE@0' ::
WINCLEARPASTE
          END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine clears the paste buffer.

# WINSTATUS

*Enables a status bar.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINSTATUS()
!MS$ ATTRIBUTES stdcall, alias:'_WINSTATUS@0' ::
WINSTATUS
         END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine enables a status bar.

# WININDEX

*Displays the index for QuickWin Help.*

### Prototype

```
      INTERFACE
       SUBROUTINE WININDEX()
!MS$ ATTRIBUTES stdcall, alias:'_WININDEX@0' ::
WININDEX
         END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine displays the index for QuickWin Help.

# WINUSING

*Displays information on how to use Help.*

### Prototype

```
       INTERFACE
        SUBROUTINE WINUSING()
!MS$ ATTRIBUTES stdcall, alias:'_WINUSING@0' ::
WINUSING
         END SUBROUTINE
       END INTERFACE
```

### Description

This callback routine displays information on how to use help.

# WINABOUT

*Displays information about the current QuickWin application.*

### Prototype

```
       INTERFACE
        SUBROUTINE WINABOUT()
!MS$ ATTRIBUTES stdcall, alias:'_WINABOUT@0' ::
WINABOUT
         END SUBROUTINE
       END INTERFACE
```

### Description

This callback routine displays information about the current QuickWin application.

# WINSELECTTEXT

*Selects text from the current window.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINSELECTTEXT()
!MS$ ATTRIBUTES stdcall, alias:'_WINSELECTTEXT@0' ::
WINSELECTTEXT
        END SUBROUTINE
      END INTERFACE
```

### Description

This callback routine selects text from the current window.

# WINSELECTGRAPHICS

*Selects graphics from the current window.*

### Prototype

```
      INTERFACE
       SUBROUTINE WINSELECTGRAPHICS()
!MS$ ATTRIBUTES stdcall,alias:'_WINSELECTGRAPHICS@0'
::  WINSELECTGRAPHICS
        END SUBROUTINE
```

```
                       END INTERFACE
```

### Description

This callback routine selects graphics from the current window.

# WINSELECTALL

*Selects the entire contents of the current window.*

### Prototype

```
          INTERFACE
           SUBROUTINE WINSELECTALL()
!MS$ ATTRIBUTES stdcall, alias:'_WINSELECTALL@0' ::
WINSELECTALL
             END SUBROUTINE
          END INTERFACE
```

### Description

This callback routine selectsthe entire contents of the current window.

# NUL

*No callback routine.*

### Prototype

```
          INTERFACE
           SUBROUTINE NUL()
!MS$ ATTRIBUTES stdcall, alias:'_NUL@0' ::  NUL
             END SUBROUTINE
```

```
END INTERFACE
```

**Description**

This is a no callback routine. It denies the use of a callback routine.

# Unknown Functions

# GETACTIVEPAGE

**Prototype**

```
INTERFACE
  FUNCTION GETACTIVEPAGE()
    INTEGER(2) GETACTIVEPAGE
  END FUNCTION
END INTERFACE
```

# GETTEXTCURSOR

**Prototype**

```
INTERFACE
  FUNCTION GETTEXTCURSOR()
    INTEGER(2) GETTEXTCURSOR
  END FUNCTION
END INTERFACE
```

# GETGTEXTVECTOR

**Prototype**

```
      INTERFACE
        SUBROUTINE GETGTEXTVECTOR(X,Y)
!MS$ ATTRIBUTES stdcall, ALIAS:"__f_getgtextvector@8"
::  getgtextvector
           INTEGER(2) X,Y
!MS$ ATTRIBUTES REFERENCE ::   X
!MS$ ATTRIBUTES REFERENCE ::   Y
        END SUBROUTINE
      END INTERFACE
```

# GETHANDLEQQ

**Prototype**

```
      INTERFACE
        FUNCTION GETHANDLEQQ(IUNIT)
          integer*4 GETHANDLEQQ, IUNIT
        END FUNCTION
      END INTERFACE
```

# GETVIDEOCONFIG

**Prototype**

```
INTERFACE
 SUBROUTINE GETVIDEOCONFIG(s)
   STRUCTURE /VIDEOCONFIG/
    INTEGER(2) NUMXPIXELS ! number of pixels on X axis
    INTEGER(2) NUMYPIXELS ! number of pixels on Y axis
    INTEGER(2) NUMTEXTCOLS ! number of text columns
                           ! available
    INTEGER(2) NUMTEXTROWS ! number of text rows
                           ! available
    INTEGER(2) NUMCOLORS   ! number of actual colors
    INTEGER(2) BITSPERPIXEL ! number of bits per pixel
    INTEGER(2) NUMVIDEOPAGES ! number of available
                             ! video pages
    INTEGER(2) MODE         ! current video mode
    INTEGER(2) ADAPTER      ! active display adapter
    INTEGER(2) MONITOR      ! active display monitor
    INTEGER(2) MEMORY       ! adapter video memory in
                            ! K bytes
   END STRUCTURE
    RECORD /VIDEOCONFIG/ S
!MS$ ATTRIBUTES REFERENCE ::  S
 END SUBROUTINE
END INTERFACE
```

# GETVISUALPAGE

### Prototype

```
INTERFACE
  FUNCTION GETVISUALPAGE()
    INTEGER(2) GETVISUALPAGE
  END FUNCTION
END INTERFACE
```

# REGISTERFONTS

### Prototype

```
INTERFACE
  FUNCTION REGISTERFONTS(FILENAME)
    INTEGER(2) REGISTERFONTS
!MS$ ATTRIBUTES ALIAS:"__ILf_registerfonts" ::
REGISTERFONTS
    CHARACTER(LEN=*) FILENAME
!MS$ ATTRIBUTES REFERENCE ::   FILENAME
  END FUNCTION
END INTERFACE
```

# SELECTPALETTE

**Prototype**

```
INTERFACE
  FUNCTION SELECTPALETTE(NUMBER)
    INTEGER(2) SELECTPALETTE,NUMBER
  END FUNCTION
END INTERFACE
```

# SETACTIVEPAGE

**Prototype**

```
INTERFACE
  FUNCTION SETACTIVEPAGE(PAGE)
    INTEGER(2) SETACTIVEPAGE,PAGE
  END FUNCTION
END INTERFACE
```

# SETFRAMEWINDOW

### Prototype

```
INTERFACE
  SUBROUTINE SETFRAMEWINDOW(X,Y,WIDTH,HEIGHT)
    INTEGER X,Y,WIDTH,HEIGHT
  END SUBROUTINE
END INTERFACE
```

# DSETGTEXTVECTOR

### Prototype

```
INTERFACE
  SUBROUTINE SETGTEXTVECTOR(X,Y)
    INTEGER(2) X,Y
  END SUBROUTINE
END INTERFACE
```

# SETSTATUSMESSAGE

### Prototype

```
INTERFACE
  SUBROUTINE SETSTATUSMESSAGE(MSG,ID)
    CHARACTER(LEN=*) MSG
!MS$ ATTRIBUTES reference ::   MSG
    INTEGER(4) ID
  END SUBROUTINE
END INTERFACE
```

# SETTEXTCURSOR

### Prototype

```
INTERFACE
  FUNCTION SETTEXTCURSOR(ATTR)
    INTEGER(2) SETTEXTCURSOR,ATTR
  END FUNCTION
END INTERFACE
```

# SETTEXTFONT

### Prototype

```
INTERFACE
  SUBROUTINE SETTEXTFONT (FONTNAME)
!MS$ ATTRIBUTES ALIAS:"__ILf_settextfont" ::
SETTEXTFONT
    character*(*) FONTNAME
!MS$ ATTRIBUTES REFERENCE ::   FONTNAME
  END SUBROUTINE
END INTERFACE
```

# SETTEXTROWS

### Prototype

```
INTERFACE
  FUNCTION SETTEXTROWS(ROWS)
    INTEGER(2) SETTEXTROWS,ROWS
  END FUNCTION
END INTERFACE
```

# SETVIDEOMODE

### Prototype

```
INTERFACE
  FUNCTION SETVIDEOMODE(MODE)
    INTEGER(2) SETVIDEOMODE,MODE
  END FUNCTION
END INTERFACE
```

# SETVIDEOMODEROWS

### Prototype

```
INTERFACE
  FUNCTION SETVIDEOMODEROWS(MODE,ROWS)
    INTEGER(2) SETVIDEOMODEROWS
    INTEGER(2) MODE,ROWS
  END FUNCTION
END INTERFACE
```

# SETVISUALPAGE

### Prototype

```
INTERFACE
  FUNCTION SETVISUALPAGE(PAGE)
    INTEGER(2) SETVISUALPAGE,PAGE
  END FUNCTION
END INTERFACE
```

# UNREGISTERFONTS

### Prototype

```
INTERFACE
  SUBROUTINE UNREGISTERFONTS()
  END SUBROUTINE
END INTERFACE
```

# Access to Windows Handles for QuickWin Components

## GETHANDLEFRAMEQQ

### Prototype

```
INTERFACE
  FUNCTION GETHANDLEFRAMEQQ()
    INTEGER(4) GETHANDLEFRAMEQQ
  END FUNCTION
END INTERFACE
```

## GETHANDLECLIENTQQ

### Prototype

```
INTERFACE
  FUNCTION GETHANDLECLIENTQQ()
    INTEGER(4) GETHANDLECLIENTQQ
  END FUNCTION
END INTERFACE
```

# GETHANDLECHILDQQ

### Prototype

```
INTERFACE
  FUNCTION GETHANDLECHILDQQ(QUICKHND)
    INTEGER(4) GETHANDLECHILDQQ
    INTEGER(4) QUICKHND
  END FUNCTION
END INTERFACE
```

# UNUSEDQQ

*Used to avoid "unused" warnings*

### Prototype

```
INTERFACE
  SUBROUTINE UNUSEDQQ()
 !MS$ ATTRIBUTES
C,REFERENCE,VARYING,ALIAS:"__FFunusedqq"::UNUSEDQQ
  END SUBROUTINE
END INTERFACE
```

### Description

Unused routine: simply returns; used to avoid "unused" warnings.

# *Index*

# H

# I

## Q

# R

## T

## U

## V-W

## X