# Fortran Library Reference

**FORTRAN 77 5.0 — Fortran 90 2.0**

901 San Antonio Road
Palo Alto, , CA 94303-4900
USA 650 960-1300 fax 650 969-9131

Part No: 805-4942
Revision A, February 1999

# Contents

# Preface

This guide describes the routines in the Sun™ FORTRAN 77 version 5.0 and Fortran 90 version 2.0 runtime libraries.

## Who Should Use This Book

This is a *reference* manual intended for programmers with a working knowledge of the Fortran language and some understanding of the Solaris™ operating environment and UNIX commands.

## Multi-Platform Release

---

**Note -** The name of the latest Solaris operating environment release is Solaris 7 but some documentation and path or package path names may still use Solaris 2.7 or SunOS 5.7.

---

The Sun™ WorkShop™ documentation applies to Solaris 2.5.1, Solaris 2.6, and Solaris 7 operating environments.

FORTRAN 77 5.0 is released for:

- The SPARC™ platform

- The x86 platform, where x86 refers to the Intel® implementation of one of the following: Intel 80386™, Intel 80486™, Pentium™, or the equivalent

Fortran 90 2.0 is released for:

- Solaris 2.5.1, 2.6, and Solaris 7 environments on SPARC processors only.

---

**Note -** The term "x86" refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture. Features described in this book that are particular to a specific platform are differentiated by the terms "SPARC" and "x86" in the text.

---

## Related Books

The following books augment this manual and provide essential information:

- *Fortran User's Guide*—provides information on command line options and how to use the compilers.
- *Fortran Programming Guide*—discusses issues relating to input/output, libraries, program analysis, debugging, performance, and so on.
- *FORTRAN 77 Language Reference*—gives details on the language.
- *Sun Performance WorkShop Fortran Overview* gives a high-level outline of the Fortran package suite.

### Other Programming Books

- *C User's Guide*—describes compiler options, pragmas, and more.
- *Numerical Computation Guide*—details floating-point computation and numerical accuracy issues.
- *Sun WorkShop Performance Library Reference*-discusses the library of subroutines and functions to perform useful operations in computational linear algebra and Fourier transforms.

### Other Sun WorkShop Books

- *Sun WorkShop Quick Install*-provides installation instructions.
- *Sun WorkShop Installation Reference*-provides supporting installation and licensing information.

- *Sun Visual WorkShop C++ Overview*-gives a high-level outline of the C++ package suite.
- *Using Sun WorkShop*—gives information on performing development operations through Sun WorkShop.
- *Debugging a Program With dbx*—provides information on using `dbx` commands to debug a program.
- *Analyzing Program Performance with Sun WorkShop*—describes the profiling tools; LoopTool, LoopReport, LockLint utilities; and the Sampling Analyzer to enhance program performance.
- *Sun WorkShop TeamWare User's Guide*—describes how to use the Sun WorkShop TeamWare code management tools.

### Solaris Books

The following Solaris manuals and guides provide additional useful information:

- *The Solaris Linker and Libraries Guide*—gives information on linking and libraries.
- The Solaris *Programming Utilities Guide*—provides information for developers about the special built-in programming tools available in the SunOS system.

## Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

## Accessing Sun Documents Online

Sun WorkShop documentation is available online from several sources:

- The `docs.sun.com` Web site
- AnswerBook2™ collections
- HTML documents
- Online help and release notes

## Using the `docs.sun.com` Web site

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

## Accessing AnswerBook2 Collections

The Sun WorkShop documentation is also available using AnswerBook2 software. To access the AnswerBook2 collections, your system administrator must have installed the AnswerBook2 documents during the installation process (if the documents are not installed, see your system administrator or Chapter 3 of *Sun WorkShop Quick Install* for installation instructions). For information about accessing AnswerBook2 documents, see Chapter 6 of *Sun WorkShop Quick Install*, Solaris installation documentation, or your system administrator.

---

**Note -** To access AnswerBook2 documents, Solaris 2.5.1 users must first download AnswerBook2 documentation server software from a Sun Web page. For more information, see Chapter 6 of *Sun WorkShop Quick Install.*

---

## Accessing HTML Documents

The following Sun Workshop documents are available online only in HTML format:

- Tools.h++ Class Library Reference
- Tools.h++ User's Guide
- *Numerical Computation Guide*
- Standard C++ Library User's Guide
- *Standard C++ Class Library Reference*
- *Sun WorkShop Performance Library Reference Manual*
- *Sun WorkShop Visual User's Guide*
- Sun WorkShop Memory Monitor User's Manual

To access these HTML documents:

1. **Open the following file through your HTML browser:**

   *install-directory*`/SUNWspro/DOC5.0/lib/locale/C/html/index.html`

Replace *install-directory* with the name of the directory where your Sun WorkShop software is installed (the default is `/opt`).

The browser displays an index of the HTML documents for the Sun WorkShop products that are installed.

2. **Open a document in the index by clicking the document's title.**

## Accessing Sun WorkShop Online Help and Release Notes

This release of Sun WorkShop includes an online help system as well as online manuals. To find out more see:

- Online Help. A help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help  Help Contents. Help menus are available in all Sun WorkShop windows.
- Release Notes. The Release Notes contain general information about Sun WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help  Release Notes.
- You can view the latest release information regarding the FORTRAN 77 and Fortran 90 compilers, by running these commands at any shell prompt:

```
% f77 -xhelp=readme -or-

% f90 -xhelp=readme
```

## What Typographic Changes Mean

The following table describes the typographic changes used in this book.

**TABLE P–1** Typographic Conventions

| Typeface or Symbol | Meaning | Example |
|---|---|---|
| `AaBbCc123` | The names of commands, files, and directories; on-screen computer output | Edit your `.login` file.<br>Use `ls -a` to list all files.<br>`machine_name% You have mail.` |
| **`AaBbCc123`** | What you type, contrasted with on-screen computer output | `machine_name%` **`su`**<br>`Password:` |
| *AaBbCc123* | Command-line placeholder:<br>replace with a real name or value | To delete a file, type `rm` *filename*. |
| *AaBbCc123* | Book titles, new words or terms, or words to be emphasized | Read Chapter 6 in *User's Guide*.<br>These are called *class* options.<br>You *must* be root to do this. |

## Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

**TABLE P–2** Shell Prompts

| Shell | Prompt |
|---|---|
| C shell prompt | `machine_name%` |
| C shell superuser prompt | `machine_name#` |
| Bourne shell and Korn shell prompt | `$` |
| Bourne shell and Korn shell superuser prompt | `#` |

## Other Conventions Used in This Book

The following conventions appear in the text of this book:

- Examples use the `csh` shell and `demo%` as the system prompt, or the `sh` shell and `demo$` as the prompt.
- Code listings and examples appear in boxes:

```
WRITE( *, * ) "Hello world"
```

- The symbol “¤” stands in for a blank space where a blank is significant:

```
¤¤36.001
```

- FORTRAN 77 examples appear in tab format, while Fortran 90 examples appear in free format. Examples common to both FORTRAN 77 and Fortran 90 use tab format except where indicated.
- Uppercase characters are generally used to show Fortran keywords and intrinsics (`PRINT`), and lowercase or mixed case is used for variables (`TbarX`).
- The Sun FORTRAN compilers are referred to by their command names, either `f77` or `f90`. "`f77/f90`" indicates information that is common to both the FORTRAN 77 and Fortran 90 compilers.
- References to online man pages appear with the topic name and section number. For example, a reference to GETENV will appear as *getenv*(3F), implying that the man command to access this page would be: `man -s 3F getenv`
- System Administrators can install the Sun Fortran compilers and supporting material at: <*install_point*>`/SUNWspro/SC5.0/` where <*install_point*> is usually `/opt` for a standard install. This is the location assumed in this book.
- The FORTRAN 77 standard uses an older convention of spelling the name "FORTRAN" capitalized. Sun documentation uses both FORTRAN and Fortran. The current convention is to use lower case: "Fortran 95".

CHAPTER **1**

# FORTRAN Library Routines

This chapter describes the Fortran library routines alphabetically. See the *FORTRAN 77 Language Reference* for details on Fortran 77 and VMS intrinsic functions. All the routines described in this chapter have corresponding man pages in section 3F of the man library. For example, **man -s 3F access** will display the man page entry for the library routine access.

## Data Type Considerations

Unless otherwise indicated, the function routines listed here are not intrinsics. That means that the type of data a function returns may conflict with the implicit typing of the function name, and require explicit type declaration by the user. For example, getpid() returns INTEGER*4 and would require an INTEGER*4 getpid declaration to ensure proper handling of the result. (Without explicit typing, a REAL result would be assumed by default because the function name starts with g.) As a reminder, explicit type statements appear in the function summaries for these routines.

Be aware that IMPLICIT statements and the -r8, -i2, -dbl and -xtypemap compiler options also alter the data typing of arguments and the treatment of return values. A mismatch between the expected and actual data types in calls to these library routines could cause unexpected behavior. Options -r8 and -dbl promote the data type of INTEGER functions to INTEGER*8, REAL functions to REAL*8, and DOUBLE functions to REAL*16. To protect against these problems, function names and variables appearing in library calls should be explicitly typed with their expected sizes, as in:

```
integer*4 seed, getuid
real*4 ran
...
```

```
    seed = 70198
    val = getuid() + ran(seed)
    ...
```

Explicit typing in the example protects the library calls from any data type promotion when the `-r8` and `-dbl` compiler options are used. Without explicit typing, these options could produce unexpected results. See the *Fortran User's Guide* and the *f77*(1) and *f90*(1) man pages for details on these options.

The more flexible `-xtypemap` compiler option is recommended over the obsolete `-i2`, `-r8`, and `-dbl` options and should be used instead.

You can catch many issues related to type mismatches over library calls by using the Fortran compilers' global program checking option, `-Xlist`. Global program checking by the `f77` and `f90` compilers is described in the *Fortran User's Guide*, the *Fortran Programming Guide*, and the *f77*(1) and *f90*(1) man pages.

## 64-Bit Environments

Compiling a program to run in a 64-bit operating environment (that is, compiling with `-xarch=v9` or `v9a` and running the executable on a SPARC platform running the 64-bit enabled Solaris 7 operating environment) changes the return values of certain functions. These are usually functions that interface standard system-level routines, such as `malloc()` (see "`malloc`, `malloc64`: Allocate Memory and Get Address" on page 69), and may take or return 32-bit or 64-bit values depending on the environment. To provide portability of code between 32-bit and 64-bit environments, 64-bit versions of these routines have been provided that always take and/or return 64-bit values. The following table identifies library routine provided for use in 64-bit environments:

**TABLE 1–1** Library Routines for 64-bit Environments

| **Library Routines** | | |
|---|---|---|
| malloc64 | Allocate memory and return a pointer | "`malloc, malloc64`: Allocate Memory and Get Address" on page 69 |
| fseeko64 | Reposition a large file | "`fseeko64, ftello64`: Determine Position and Reposition a Large File " on page 23 |
| ftello64 | Determine position of a large file | "`fseeko64, ftello64`: Determine Position and Reposition a Large File " on page 23 |
| stat64, fstat64, lstat64 | Determine status of a file | "`stat64, lstat64, fstat64`: Get File Status " on page 88 |
| time64, ctime64, gmtime64, ltime64 | Get system time, convert to character or dissected | "`time`, `ctime`, `ltime`, `gmtime`: Get System Time" on page 90 |
| qsort64 | Sort the elements of an array | "`qsort,qsort64`: Sort the Elements of a One-dimensional Array " on page 76 |

## `abort`: Terminate and Write Core File

The subroutine is called by:

```
call abort
```

`abort` flushes the I/O buffers and then aborts the process, possibly producing a `core` file memory dump in the current directory. See *limit*(1) about limiting or suppressing core dumps.

## `access`: Check File Permissions or Existence

The function is called by:

| `INTEGER*4 access`<br>`status = access (` *name, mode* `)` | | | |
|---|---|---|---|
| *name* | `character` | Input | File name |
| *mode* | `character` | Input | Permissions |
| Return value | `INTEGER*4` | Output | *status*=0: OK *status*>0: Error code |

`access` determines if you can access the file *name* with the permissions specified by *mode*. access returns zero if the access specified by *mode* would be successful. See also *gerror*(3F) to interpret error codes.

Set *mode* to one or more of `r`, `w`, or `x,` in any order or combination, where `r`, `w`, `x` have the following meanings:

|  |  |
|---|---|
| r | Test for read permission |
| w | Test for write permission |
| x | Test for execute permission |
| blank | Test for existence of the file |

Example 1: Test for read / write permission:

```
INTEGER*4  access, status
status = access ( "taccess.data", "rw" )
if ( status .eq. 0 ) write(*,*) "ok"
if ( status .ne. 0 ) write(*,*) "cannot read/write", status
```

Example 2: Test for existence:

```
INTEGER*4  access, status
status = access ( "taccess.data", " " )    ! blank mode
if ( status .eq. 0 ) write(*,*) "file exists"
if ( status .ne. 0 ) write(*,*) "no such file", status
```

## `alarm`: Call Subroutine After a Specified Time

The function is called by:

| INTEGER*4 alarm<br><br>n = alarm ( ***time***, ***sbrtn*** ) |  |  |  |
|---|---|---|---|
| *time* | INTEGER*4 | Input | Number of seconds to wait (0=do not call) |
| *sbrtn* | Routine name | Input | Subprogram to execute must be listed in an external statement. |
| Return value | INTEGER*4 | Output | Time remaining on the last alarm |

Example: `alarm`—wait 9 seconds then call `sbrtn`:

```
integer*4 alarm, time / 1 /
common / alarmcom / i
external sbrtn
i = 9
write(*,*) i
nseconds =  alarm ( time, sbrtn )
do n = 1,100000         ! Wait until alarm activates sbrtn.
     r = n              ! (any calculations that take enough time)
   x=sqrt(r)
end do
write(*,*) i
end
subroutine sbrtn
common / alarmcom / i
i = 3                    ! Do no I/O in this routine.
return
end
```

See also: *alarm*(3C), *sleep*(3F), and *signal*(3F). Note the following restrictions:

- A subroutine cannot pass its own name to `alarm`.
- The `alarm` routine generates signals that could interfere with any I/O. The called subroutine, *sbrtn*, must not do any I/O itself.
- Calling `alarm()` from a parallelized or multi-threaded FORTRAN program may have unpredictable results.

## `bit`: Bit Functions: `and`, `or`, …, `bit`, `setbit`, …

The definitions are:

| | |
|---|---|
| `and(` *word1*, *word2* `)` | Computes bitwise *and* of its arguments. |
| `or(` *word1*, *word2* `)` | Computes bitwise *inclusive or* of its arguments. |
| `xor(` *word1*, *word2* `)` | Computes bitwise *exclusive or* of its arguments. |
| `not(` *word* `)` | Returns bitwise *complement* of its argument. |
| `lshift(` *word*, *nbits* `)` | Logical left shift with no end around carry. |

| | |
|---|---|
| `rshift(` *word, nbits* `)` | Arithmetic right shift with sign extension. |
| `call bis(` *bitnum*, *word* `)` | Sets bit *bitnum* in *word* to 1. |
| `call bic(` *bitnum*, *word* `)` | Clears bit *bitnum* in *word* to 0. |
| `bit(` *bitnum*, *word* `)` | Tests bit *bitnum* in *word* and returns `.true.` if the bit is 1, `.false.` if it is 0. |
| `call setbit(`*bitnum*,*word*,*state*`)` | Sets bit *bitnum* in *word* to 1 if *state* is nonzero, and clears it otherwise. |

The alternate external versions for MIL-STD-1753 are:

| | |
|---|---|
| `iand(` *m*, *n* `)` | Computes the bitwise *and* of its arguments. |
| `ior(` *m*, *n* `)` | Computes the bitwise *inclusive or* of its arguments. |
| `ieor(` *m*, *n* `)` | Computes the bitwise *exclusive or* of its arguments. |
| `ishft(` *m*, *k* `)` | Is a logical shift with no end around carry (left if *k*>0, right if *k*<0). |
| `ishftc(` *m*, *k*, *ic* `)` | Circular shift: right-most *ic* bits of *m* are left-shifted circularly *k* places. |
| `ibits(` *m*, *i*, *len* `)` | Extracts bits: from *m*, starting at bit *i*, extracts *len* bits. |
| `ibset(` *m*, *i* `)` | Sets bit: return value is equal to word *m* with bit number *i* set to 1. |
| `ibclr(` *m*, *i* `)` | Clears bit: return value is equal to word *m* with bit number *i* set to 0. |
| `btest(` *m*, *i* `)` | Tests bit *i* in *m*; returns `.true.` if the bit is 1, and `.false.` if it is 0. |

See also “`mvbits`: Move a Bit Field ” on page 71, and the chapter on Intrinsic Functions in the *FORTRAN 77 Reference Manual*.

## Usage: `and`, `or`, `xor`, `not`, `rshift`, `lshift`

For the intrinsic functions:

*x* = `and(` *word1*, *word2* `)`

*x* = `or(` *word1*, *word2* `)`

*x* = `xor(` *word1*, *word2* `)`

*x* = `not(` *word* `)`

*x* = `rshift(` *word*, *nbits* `)`

*x* = `lshift(` *word*, *nbits* `)`

*word*, *word1*, *word2*, *nbits* are integer input arguments. These are intrinsic functions expanded inline by the compiler. The data type returned is that of the first argument.

No test is made for a reasonable value of *nbits*.

Example: `and, or, xor, not`:

```
demo% cat tandornot.f
    print 1, and(7,4), or(7,4), xor(7,4), not(4)
 1    format(4x "and(7,4)", 5x "or(7,4)", 4x "xor(7,4)",
&          6x "not(4)"/4o12.11)
    end
demo% f77 -silent tandornot.f
demo% a.out
    and(7,4)     or(7,4)    xor(7,4)      not(4)
 00000000004 00000000007 00000000003 37777777773
demo%
```

Example: `lshift`, `rshift`:

```
    integer*4 lshift, rshift
    print 1, lshift(7,1), rshift(4,1)
 1    format(1x "lshift(7,1)", 1x "rshift(4,1)"/2o12.11)
    end
demo% f77 -silent tlrshift.f
demo% a.out
 lshift(7,1) rshift(4,1)
 00000000016 00000000002
demo%
```

## Usage: `bic`, `bis`, `bit`, `setbit`

`call bic(` *bitnum*, *word* `)`

`call bis(` *bitnum*, *word* `)`

`call setbit(` *bitnum*, *word*, *state* `)`

`LOGICAL bit` *x* = `bit(` *bitnum*, *word* `)`

*bitnum*, *state*, and *word* are `INTEGER*4` input arguments. Function bit() returns a logical value.

Bits are numbered so that bit 0 is the least significant bit, and bit 31 is the most significant.

`bic`, `bis`, and `setbit` are external subroutines. `bit` is an external function.

Example 3: `bic`, `bis`, `setbit`, `bit`:

```
    integer*4 bitnum/2/, state/0/, word/7/
    logical bit
    print 1, word
 1    format(13x "word", o12.11)
    call bic( bitnum, word )
    print 2, word
 2    format("after bic(2,word)", o12.11)
    call bis( bitnum, word )
    print 3, word
 3    format("after bis(2,word)", o12.11)
    call setbit( bitnum, word, state )
    print 4, word
 4    format("after setbit(2,word,0)", o12.11)
    print 5, bit(bitnum, word)
 5    format("bit(2,word)", L )
    end
```

*<output>*

```
             word 00000000007
after bic(2,word) 00000000003
after bis(2,word) 00000000007
after setbit(2,word,0) 00000000003
bit(2,word) F
```

## `chdir`: Change Default Directory

The function is called by:

| INTEGER*4 chdir<br><br>***n*** = chdir( ***dirname*** ) | | | |
|---|---|---|---|
| *dirname* | character | Input | Directory name |
| Return value | INTEGER*4 | Output | *n*=0: OK, *n*>0: Error code |

Example: chdir—change cwd to MyDir:

```
INTEGER*4  chdir, n
n =  chdir ( "MyDir" )
if ( n .ne. 0 ) stop "chdir: error"
end
```

See also: *chdir*(2), *cd*(1), and *gerror*(3F) to interpret error codes.

Path names can be no longer than MAXPATHLEN as defined in <sys/param.h>. They can be relative or absolute paths.

Use of this function can cause inquire by unit to fail.

Certain FORTRAN file operations reopen files by name. Using chdir while doing I/O can cause the runtime system to lose track of files created with relative path names. including the files that are created by open statements without file names.

## chmod: Change the Mode of a File

The function is called by:

| `INTEGER*4 chmod`<br><br>***n*** = `chmod(` ***name***, ***mode*** `)` | | | |
|---|---|---|---|
| *name* | `character` | Input | Path name |
| *mode* | `character` | Input | Anything recognized by *chmod*(1), such as `o-w`, `444`, etc. |
| Return value | `INTEGER*4` | Output | *n* = 0: OK; *n*>0: System error number |

Example: `chmod`—add write permissions to `MyFile`:

```
character*18 name, mode
INTEGER*4 chmod, n
name = "MyFile"
mode = "+w"
n =  chmod( name, mode )
if ( n .ne. 0 ) stop ’chmod: error’
end
```

See also: *chmod*(1), and *gerror*(3F) to interpret error codes.

Path names cannot be longer than `MAXPATHLEN` as defined in `<sys/param.h>`. They can be relative or absolute paths.

## `date`: Get Current Date as a Character String

**Note -** This routine is not “Year 2000 Safe” because it returns only a two-digit value for the year. Programs that compute differences between dates using the output of this routine may not work properly after 31 December, 1999. Programs using this `date()` routine will see a runtime warning message the first time the routine is called to alert the user. See `date_and_time()` as a possible alternate routine.

The subroutine is called by:

| call date( *c* ) | | |
|---|---|---|
| *c* CHARACTER*9 | Output | Variable, array, array element, or character substring |

The form of the returned string *c* is *dd-mmm-yy*, where *dd* is the day of the month as a 2-digit number, *mmm* is the month as a 3-letter abbreviation, and *yy* is the year as a 2-digit number (and is not year 2000 safe!).

Example: `date`:

```
demo% cat dat1.f
* dat1.f -- Get the date as a character string.
    character c*9
    call date ( c )
    write(*,"(" The date today is: ", A9 )" ) c
    end
demo% f77 -silent dat1.f
   "dat.f", line 2: Warning: Subroutine "date" is not safe after
       year 2000; use "date_and_time" instead
demo% a.out
Computing time differences using the 2 digit year from subroutine
       date is not safe after year 2000.
 The date today is: 9-Jul-98
demo%
```

See also `idate()` and `date_and_time()`.

## `date_and_time`: Get Date and Time

This is a FORTRAN 77 version of the Fortran 90 intrinsic routine, and is Year 2000 safe.

The `date_and_time` subroutine returns data from the real-time clock and the date. Local time is returned, as well as the difference between local time and Universal Coordinated Time (UTC) (also known as Greenwich Mean Time, GMT).

The `date_and_time`() subroutine is called by:

| `call date_and_time(` ***date, time, zone, values*** `)` | | | |
|---|---|---|---|
| *date* | `CHARACTER*8` | Output | Date, in form CCYYMMDD, where CCYY is the four-digit year, MM the two-digit month, and DD the two-digit day of the month. For example: 19980709 |
| *time* | `CHARACTER*10` | Output | The current time, in the form hhmmss.sss, where hh is the hour, mm minutes, and ss.sss seconds and milliseconds. |
| *zone* | `CHARACTER*5` | Output | The time difference with respect to UTC, expressed in hours and minutes, in the form hhmm |
| *values* | `INTEGER*4 VALUES(8)` | Output | An integer array of 8 elements described below. |

The eight values returned in the `INTEGER*4` *values* array are

| | |
|---|---|
| `VALUES(1)` | The year, as a 4-digit integer. For example, 1998. |
| `VALUES(2)` | The month, as an integer from 1 to 12. |
| `VALUES(3)` | The day of the month, as an integer from 1 to 31. |
| `VALUES(4)` | The time difference, in minutes, with respect to UTC. |
| `VALUES(5)` | The hour of the day, as an integer from 1 to 23. |
| `VALUES(6)` | The minutes of the hour, as an integer from 1 to 59. |
| `VALUES(7)` | The seconds of the minute, as an integer from 0 to 60. |
| `VALUES(8)` | The milliseconds of the second, in range 0 to 999. |

An example using date_and_time:

```
demo% cat dtm.f
       integer date_time(8)
```

```
        character*10 b(3)
        call date_and_time(b(1), b(2), b(3), date_time)
        print *,"date_time    array values:"
        print *,"year=",date_time(1)
        print *,"month_of_year=",date_time(2)
        print *,"day_of_month=",date_time(3)
        print *,"time difference in minutes=",date_time(4)
        print *,"hour of day=",date_time(5)
        print *,"minutes of hour=",date_time(6)
        print *,"seconds of minute=",date_time(7)
        print *,"milliseconds of second=",date_time(8)
        print *, "DATE=",b(1)
        print *, "TIME=",b(2)
        print *, "ZONE=",b(3)
        end
```

When run on a computer in California, USA on July 9, 1998, it generated the following output:

```
 date_time array values:
 year=  1998
 month_of_year=  7
 day_of_month=  9
 time difference in minutes=  -420
 hour of day=  17
 minutes of hour=  8
 seconds of minute=  54
 milliseconds of second=  587
 DATE=19980709
 TIME=170854.587
 ZONE=-0700
```

## `dtime`, `etime`: Elapsed Execution Time

Both functions have return values of elapsed time (or -1.0 as error indicator). The time is in seconds. The resolution is to a nanosecond.

### `dtime`: Elapsed Time Since the Last `dtime` Call

For `dtime`, the elapsed time is:

- First call: elapsed time since start of execution
- Subsequent calls: elapsed time since the last call to `dtime`
- Single processor: time used by the CPU

- Multiple Processor: the sum of times for all the CPUs, which is not useful data; use `etime` instead.

---

**Note -** Do not call `dtime` from within a parallelized loop.

---

The function is called by:

***e*** = `dtime(` ***tarray*** `)`

| | | | |
|---|---|---|---|
| *tarray* | `real(2)` | Output | *e*= -1.0: Error: *tarray* values are undefined<br>*e*¬= -1.0: User time in *tarray(1)* if no error. System time in *tarray(2)* if no error |
| Return value | `real` | Output | *e*= -1.0: Error<br>*e*¬= -1.0: The sum of *tarray(1)* and *tarray(2)* |

Example: `dtime()`, single processor:

```
    real e, dtime, t(2)
    print *, "elapsed:", e, ", user:", t(1), ", sys:", t(2)
    do i = 1, 10000
        k=k+1
    end do
    e = dtime( t )
    print *, "elapsed:", e, ", user:", t(1), ", sys:", t(2)
    end
demo% f77 -silent tdtime.f
demo% a.out
elapsed:  0., user:  0., sys:  0.
elapsed:   0.180000, user:    6.00000E-02, sys:   0.120000
demo%
```

## `etime`: Elapsed Time Since Start of Execution

For `etime`, the elapsed time is:

- Single Processor-CPU time for the calling process
- Multiple Processors—wallclock time while processing your program

Here is how FORTRAN decides single processor or multiple processor:

For a parallelized FORTRAN program linked with `libF77_mt`, if the environment variable `PARALLEL` is:

- Undefined, the current run is single processor.

- Defined and in the range 1, 2, 3, …, the current run is multiple processor.
- Defined, but some value other than 1, 2, 3, …, the results are unpredictable.

The function is called by:

| ***e*** = `etime(` ***tarray*** `)` | | | |
|---|---|---|---|
| *tarray* | `real(2)` | Output | *e*= -1.0: Error: *tarray* values are undefined.<br><br>*e*¬= -1.0: Single Processor: User time in *tarray(1)*. System time in *tarray(2)*<br><br>Multiple Processor: Wall clock time in *tarray(1)*, 0.0 in *tarray(2)* |
| Return value | `real` | Output | *e*= -1.0: Error<br><br>*e*¬= -1.0: The sum of *tarray(1)* and *tarray(2)* |

Take note that the initial call to etime will be inaccurate. It merely enables the system clock. Do not use the value returned by the initial call to etime.

Example: `etime()`, single processor:

```
    real e, etime, t(2)
    e = etime(t)         !  Startup etime - do not use result
    do i = 1, 10000
        k=k+1
    end do
    e = etime( t )
    print *, "elapsed:", e, ", user:", t(1), ‘, sys:", t(2)
    end
demo% f77 -silent tetime.f
demo% a.out
elapsed:   0.190000, user:    6.00000E-02, sys:   0.130000
demo%
```

See also *times*(2), *f77*(1), and the *Fortran Programming Guide*.

## `exit`: Terminate a Process and Set the Status

The subroutine is called by:

| `call exit(` *status* `)` | | |
|---|---|---|
| *status* | `INTEGER*4` | Input |

Example: `exit()`:

```
     ...
    if(dx .lt. 0.) call exit( 0 )
    ...
    end
```

`exit` flushes and closes all the files in the process, and notifies the parent process if it is executing a `wait`.

The low-order 8 bits of *status* are available to the parent process. These 8 bits are shifted left 8 bits, and all other bits are zero. (Therefore, *status* should be in the range of 256 - 65280). This call will never return.

The C function `exit` can cause cleanup actions before the final system "`exit`".

Calling `exit` without an argument causes a compile-time warning message, and a zero will be automatically provided as an argument. See also: *exit*(2), *fork*(2), *fork*(3F), *wait*(2), *wait*(3F).

## `fdate`: Return Date and Time in an ASCII String

The subroutine or function is called by:

| `call fdate(` ***string*** `)` | | |
|---|---|---|
| *string* | `character*24` | Output |

or:

| | | | |
|---|---|---|---|
| `CHARACTER fdate*24`<br><br>*string* = `fdate()` | | | If used as a function, the calling routine must define the type and size of `fdate`. |
| Return value | `character*24` | Output | |

Example 1: `fdate` as a subroutine:

```
    character*24 string
    call fdate( string )
    write(*,*) string
    end
```

Output:

```
 Wed Aug  3 15:30:23 1994
```

Example 2: `fdate` as a function, same output:

```
    character*24 fdate
    write(*,*)  fdate()
    end
```

See also: *ctime*(3), *time*(3F), and *idate*(3F).

## `flush`: Flush Output to a Logical Unit

The subroutine is called by:

| `call flush(` *lunit* `)` | | | |
|---|---|---|---|
| *lunit* | `INTEGER*4` | Input | Logical unit |

The `flush` subroutine flushes the contents of the buffer for the logical unit, `lunit`, to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the console terminal.

See also *fclose*(3S).

## `fork`: Create a Copy of the Current Process

The function is called by:

| `INTEGER*4 fork`<br><br>***n*** = `fork()` | | | |
|---|---|---|---|
| Return value | `INTEGER*4` | Output | *n*>0: *n*=Process ID of copy<br><br>*n*<0, *n*=System error code |

The `fork` function creates a copy of the calling process. The only distinction between the two processes is that the value returned to one of them, referred to as the *parent* process, will be the process ID of the copy. The copy is usually referred to as the *child* process. The value returned to the child process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external files.

Example: `fork()`:

```
INTEGER*4 fork, pid
pid = fork()
if(pid.lt.0) stop "fork error"
if(pid.gt.0) then
    print *, "I am the parent"
else
    print *, "I am the child"
endif
```

A corresponding `exec` routine has not been provided because there is no satisfactory way to retain open logical units across the `exec` routine. However, the usual function of `fork/exec` can be performed using *system*(3F). See also: *fork*(2), *wait*(3F), *kill*(3F), *system*(3F), and *perror*(3F).

## `free`: Deallocate Memory Allocated by Malloc

The subroutine is called by:

| call free ( ***ptr*** ) | | |
|---|---|---|
| *ptr* | pointer | Input |

`free` deallocates a region of memory previously allocated by `malloc`. The region of memory is returned to the memory manager; it is no longer available to the user's program.

Example: `free()`:

```
real x
pointer ( ptr, x )
ptr = malloc ( 10000 )
call free ( ptr )
end
```

See "`malloc`, `malloc64`: Allocate Memory and Get Address" on page 69 for details.

## `fseek`, `ftell`: Determine Position and Reposition a File

`fseek` and `ftell` are routines that permit repositioning of a file. `ftell` returns a file's current position as an offset of so many bytes from the beginning of the file. At some later point in the program, `fseek` can use this saved offset value to reposition the file to that same place for reading.

### `fseek`: Reposition a File on a Logical Unit

The function is called by:

| `INTEGER*4 fseek`<br><br>***n*** = `fseek(` ***lunit***, ***offset***, ***from*** `)` | | | |
|---|---|---|---|
| *lunit* | `INTEGER*4` | Input | Open logical unit |
| *offset* | `INTEGER*4`<br>*or*<br>`INTEGER*8` | Input | Offset in bytes relative to position specified by *from* |
| | An `INTEGER*8` offset value is required when compiled for a 64-bit environment, such as Solaris 7, with `-xarch=v9`. If a literal constant is supplied, it must be a 64-bit constant, for example: `100_8` | | |
| *from* | `INTEGER*4` | Input | 0=Beginning of file<br>1=Current position<br>2=End of file |
| Return value | `INTEGER*4` | Output | *n*=0: OK; *n*>0: System error code |

**Note -** On sequential files, following a call to `fseek` by an output operation (e.g. WRITE) causes all data records following the `fseek` position to be deleted and replaced by the new data record (and an end-of-file mark). Rewriting a record in place can only be done with direct access files.

Example: `fseek()`—Reposition `MyFile` to two bytes from the beginning

```
INTEGER*4 fseek, lunit/1/, offset/2/, from/0/, n
open( UNIT=lunit, FILE="MyFile" )
n = fseek( lunit, offset, from )
if ( n .gt. 0 ) stop "fseek error"
end
```

:

Example: Same example in a 64-bit environment and compiled with `-xarch=v9`:

```
INTEGER*4 fseek, lunit/1/,  from/0/, n
INTEGER*8 offset/2/
open( UNIT=lunit, FILE="MyFile" )
n = fseek( lunit, offset, from )
if ( n .gt. 0 ) stop "fseek error"
end
```

## `ftell`: Return Current Position of File

The function is called by:

| `INTEGER*4 ftell`<br><br>***n*** = `ftell(` ***lunit*** `)` | | | |
|---|---|---|---|
| *lunit* | `INTEGER*4` | Input | Open logical unit |
| Return value | `INTEGER*4`<br>*or*<br>`INTEGER*8` | Output | *n*>=0: *n*=Offset in bytes from start of file<br>*n*<0: *n*=System error code |
| | An `INTEGER*8` offset value is returned when compiling for a 64-bit environment, such as Solaris 7, with `-xarch=v9`. `ftell` and variables receiving this return value should be declared `INTEGER*8`. | | |

Example: `ftell()`:

```
INTEGER*4 ftell, lunit/1/, n
open( UNIT=lunit, FILE="MyFile" )
...
n = ftell( lunit )
if ( n .lt. 0 ) stop "ftell error"
...
```

Example: Same example in a 64-bit environment and compiled with `-xarch=v9`:

```
INTEGER*4 lunit/1/
INTEGER*8 ftell, n
open( UNIT=lunit, FILE="MyFile" )
...
n = ftell( lunit )
if ( n .lt. 0 ) stop "ftell error"
...
```

See also *fseek*(3S) and *perror*(3F); also *fseeko64*(3F) *ftello64*(3F).

## `fseeko64`, `ftello64`: Determine Position and Reposition a Large File

`fseeko64` and `ftello64` are "large file" versions of fseek and ftell. They take and return INTEGER*8 file position offsets on Solaris 2.6 and Solaris 7. (A "large file" is larger than 2 Gigabytes and therefore a byte-position must be represented by a 64-bit integer.) Use these versions to determine and/or reposition large files.

### `fseeko64`: Reposition a File on a Logical Unit

The function is called by:

| INTEGER fseeko64<br><br>***n*** = fseeko64( ***lunit***, ***offset64***, ***from*** ) | | | |
|---|---|---|---|
| *lunit* | `INTEGER*4` | Input | Open logical unit |
| *offset64* | `INTEGER*8` | Input | 64-bit offset in bytes relative to position specified by *from* |
| *from* | `INTEGER*4` | Input | 0=Beginning of file<br>1=Current position<br>2=End of file |
| Return value | `INTEGER*4` | Output | *n*=0: OK; *n*>0: System error code |

---

**Note -** On sequential files, following a call to `fseeko64` by an output operation (e.g. WRITE) causes all data records following the `fseek` position to be deleted and replaced by the new data record (and an end-of-file mark). Rewriting a record in place can only be done with direct access files.

---

Example: `fseeko64()`—Reposition `MyFile` to two bytes from the beginning:

```
INTEGER fseeko64, lunit/1/, from/0/, n
INTEGER*8 offset/200/
open( UNIT=lunit, FILE="MyFile" )
n = fseeko64( lunit, offset, from )
if ( n .gt. 0 ) stop "fseek error"
end
```

## `ftello64`: Return Current Position of File

The function is called by:

| `INTEGER*8 ftello64`<br><br>***n*** = `ftello64(` ***lunit*** `)` | | | |
|---|---|---|---|
| *lunit* | `INTEGER*4` | Input | Open logical unit |
| Return value | `INTEGER*8` | Output | *n*>=0: *n*=Offset in bytes from start of file<br><br>*n*<0: *n*=System error code |

Example: `ftello64()`:

```
INTEGER*8 ftello64, lunit/1/, n
open( UNIT=lunit, FILE="MyFile" )
...
n = ftello64( lunit )
if ( n .lt. 0 ) stop "ftell error"
...
```

# `getarg`, `iargc`: Get Command-line Arguments

`getarg` and `iargc` access arguments on the command line (after expansion by the command-line preprocessor.

## `getarg`: Get a Command-Line Argument

The subroutine is called by:

| call getarg( ***k***, ***arg*** ) | | |
|---|---|---|
| *k* INTEGER*4 | Input | Index of argument (0=first=command name) |
| *arg* character**n* | Output | *k*th argument |
| *n* INTEGER*4 | Size of *arg* | Large enough to hold longest argument |

## iargc: Get the Number of Command-Line Arguments

The function is called by:

| ***m*** = iargc() | | | |
|---|---|---|---|
| Return value | INTEGER*4 | Output | Number of arguments on the command line |

Example: iargc and getarg, get argument count and each argument:

```
demo% cat yarg.f
    character argv*10
    INTEGER*4 i, iargc, n
    n = iargc()
    do 1 i = 1, n
     call getarg( i, argv )
 1   write( *, "( i2, 1x, a )" ) i, argv
    end
demo% f77 -silent yarg.f
demo% a.out *.f
1 first.f
2 yarg.f
```

See also *execve*(2) and *getenv*(3F).

## `getc`, `fgetc`: Get Next Character

`getc` and `fgetc` get the next character from the input stream.Do not mix calls to these routines with normal Fortran I/O on the same logical unit.

### `getc`: Get Next Character from `stdin`

The function is called by:

| INTEGER*4 getc<br><br>***status*** = getc( ***char*** ) | | | |
|---|---|---|---|
| *char* | `character` | Output | Next character |
| Return value | `INTEGER*4` | Output | *status*=0: OK<br>*status*=-1: End of file<br>*status*>0: System error code or `f77` I/O error code |

Example: `getc` gets each character from the keyboard; note the Control-D (`^D`):

```
    character char
    INTEGER*4 getc, status
    status = 0
    do while ( status .eq. 0 )
        status = getc( char )
        write(*, "(i3, o4.3)") status, char
    end do
    end
```

After compiling, a sample run of the above source is:

```
demo% a.out
ab              Program reads letters typed in
                   terminated by a CONTROL-D.
^D
0 141           Program outputs status and octal value of the characters entered
0 142              141 represents 'a', 142 is 'b'
0 012              012 represents the RETURN key
-1 012          Next attempt to read returns CONTROL-D
demo%
```

For any logical unit, do not mix normal FORTRAN input with `getc()`.

## `fgetc`: Get Next Character from Specified Logical Unit

The function is called by:

| `INTEGER*4 fgetc`<br><br>***status*** = `fgetc(` ***lunit***, ***char*** `)` | | | |
|---|---|---|---|
| *lunit* | `INTEGER*4` | Input | Logical unit |
| *char* | `character` | Output | Next character |
| Return value | `INTEGER*4` | Output | *status*=-1: End of File<br><br>*status*>0: System error code or `f77` I/O error code |

Example: `fgetc` gets each character from `tfgetc.data`; note the linefeeds (Octal 012):

```
    character char
    INTEGER*4 fgetc, status
    open( unit=1, file="tfgetc.data" )
    status = 0
    do while ( status .eq. 0 )
        status = fgetc( 1, char )
        write(*, "(i3, o4.3)") status, char
    end do
    end
```

After compiling, a sample run of the above source is:

```
demo% cat tfgetc.data
ab
yz
demo% a.out
0 141         'a' read
0 142         'b' read
0 012         linefeed read
```

```
0 171        'y'  read
0 172        'z'  read
0 012        linefeed  read
-1 012       CONTROL-D  read
demo%
```

For any logical unit, do not mix normal FORTRAN input with `fgetc()`.

See also: *getc*(3S), *intro*(2), and *perror*(3F).

## `getcwd`: Get Path of Current Working Directory

The function is called by:

| INTEGER*4 getcwd<br><br>***status*** = getcwd( ***dirname*** ) | | | |
|---|---|---|---|
| *dirname* | `character*`*n* | Output<br>The path of the current directory is returned | Path name of the current working directory. *n* must be large enough for longest path name |
| Return value | `INTEGER*4` | Output | *status*=0: OK<br><br>*status*>0: Error code |

Example: `getcwd`:

```
    INTEGER*4 getcwd, status
    character*64 dirname
    status = getcwd( dirname )
    if ( status .ne. 0 ) stop "getcwd: error"
    write(*,*) dirname
    end
```

See also: *chdir*(3F), *perror*(3F), and *getwd*(3).

Note: the path names cannot be longer than `MAXPATHLEN` as defined in `<sys/param.h>`.

## `getenv`: Get Value of Environment Variables

The subroutine is called by:

| `call getenv(` ***ename***, ***evalue*** `)` | | | |
|---|---|---|---|
| *ename* | `character*`*n* | Input | Name of the environment variable sought |
| *evalue* | `character*`*n* | Output | Value of the environment variable found; blanks if not successful |

The size of *ename* and *evalue* must be large enough to hold their respective character strings.

The `getenv` subroutine searches the environment list for a string of the form *ename=evalue* and returns the value in *evalue* if such a string is present; otherwise, it fills *evalue* with blanks.

Example: Use `getenv()` to print the value of `$SHELL`:

```
    character*18  evalue
    call getenv( "SHELL", evalue )
    write(*,*) "'", evalue, "'"
    end
```

See also: *execve*(2) and *environ*(5).

## `getfd`: Get File Descriptor for External Unit Number

The function is called by:

| INTEGER*4 getfd<br><br>***fildes*** = getfd( ***unitn*** ) | | | |
|---|---|---|---|
| *unitn* | INTEGER*4 | Input | External unit number |
| Return value | INTEGER*4 *-or-* INTEGER*8 | Output | File descriptor if file is connected; -1 if file is not connected An INTEGER*8 result is returned when compiling for 64-bit environments |

Example: getfd():

```
INTEGER*4 fildes, getfd, unitn/1/
open( unitn, file="tgetfd.data" )
fildes = getfd( unitn )
if ( fildes .eq. -1 ) stop "getfd: file not connected"
write(*,*) "file descriptor = ", fildes
end
```

See also *open*(2).

## getfilep: Get File Pointer for External Unit Number

The function is:

| ***irtn*** = ***c_read***( getfilep( ***unitn*** ), ***inbyte***, 1 ) | | | |
|---|---|---|---|
| *c_read* | C function | Input | User's own C function. See example. |
| *unitn* | INTEGER*4 | Input | External unit number. |
| getfilep | INTEGER*4 *-or-* INTEGER*8 | Return value | File pointer if the file is connected; -1 if the file is not connected. An INTEGER*8 value is returned when compiling for 64-bit environments |

This function is used for mixing standard FORTRAN I/O with C I/O. Such a mix is nonportable, and is not guaranteed for subsequent releases of the operating system or FORTRAN. Use of this function is not recommended, and no direct interface is provided. You must create your own C routine to use the value returned by `getfilep`. A sample C routine is shown below.

Example: FORTRAN uses `getfilep` by passing it to a C function:

`tgetfilepF.f:`

```
      character*1  inbyte
      integer*4    c_read,  getfilep, unitn / 5 /
      external     getfilep
      write(*,"(a,$)") "What is the digit? "

      irtn = c_read( getfilep( unitn ), inbyte, 1 )

      write(*,9)  inbyte
    9 format("The digit read by C is ", a )
      end
```

Sample C function actually using `getfilep`:

`tgetfilepC.c:`

```
      #include <stdio.h>
      int c_read_ ( fd, buf, nbytes, buf_len )
      FILE **fd ;
      char *buf ;
      int *nbytes, buf_len ;
      {
         return fread( buf, 1, *nbytes, *fd ) ;
      }
```

A sample compile-build-run is:

```
demo 11% cc -c tgetfilepC.c
demo 12% f77 tgetfilepC.o tgetfilepF.f
tgetfileF.f:
MAIN:
demo 13% a.out
What is the digit? 3
The digit read by C is 3
demo 14%
```

For more information, read the chapter on the C-FORTRAN interface in the *Fortran Programming Guide*. See also *open*(2).

## getlog: Get User's Login Name

The subroutine is called by:

| call getlog( ***name*** ) | | | |
|---|---|---|---|
| *name* | character**n* | Output | User's login name, or all blanks if the process is running detached from a terminal. *n* should be large enough to hold the longest name. |

Example: getlog:

```
character*18 name
call getlog( name )
write(*,*) "'", name, "'"
end
```

See also *getlogin*(3).

## getpid: Get Process ID

The function is called by:

| INTEGER*4 getpid<br><br>***pid*** = getpid() | | | |
|---|---|---|---|
| Return value | INTEGER*4 | Output | Process ID of the current process |

Example: getpid:

```
INTEGER*4 getpid, pid
pid = getpid()
write(*,*) "process id = ", pid
end
```

See also *getpid*(2).

## getuid, getgid: Get User or Group ID of Process

getuid and getgid get the user or group ID of the process, respectively.

### getuid: Get User ID of the Process

The function is called by:

| INTEGER*4 getuid<br>***uid* = getuid()** | | | |
|---|---|---|---|
| Return value | INTEGER*4 | Output | User ID of the process |

### getgid: Get Group ID of the Process

The function is called by:

| INTEGER*4 getgid<br>gid = getgid() | | | |
|---|---|---|---|
| Return value | INTEGER*4 | Output | Group ID of the process |

Example: getuid() and getpid():

```
INTEGER*4 getuid, getgid, gid, uid
uid = getuid()
gid = getgid()
write(*,*) uid, gid
end
```

See also: *getuid*(2).

## `hostnm`: Get Name of Current Host

The function is called by:

| `INTEGER*4 hostnm`<br>***status*** = `hostnm(` ***name*** `)` | | | |
|---|---|---|---|
| *name* | `character*`*n* | Output | Name of current host system. *n* must be large enough to hold the host name. |
| Return value | `INTEGER*4` | Output | *status*=0: OK<br>*status*>0: Error |

Example: `hostnm()`:

```
INTEGER*4 hostnm, status
character*8 name
status = hostnm( name )
write(*,*) "host name = "", name, """
end
```

See also *gethostname*(2).

## `idate`: Return Current Date

`idate` has two versions:

- *Standard*—Put the current system date into an integer array: day, month, and year.
- *VMS*—Put the current system date into three integer variables: month, day, and year. This version is not “Year 2000 Safe”.

The `-lV77` compiler option request the VMS library and links the VMS versions of both `time()` and `idate()`; otherwise, the linker accesses the standard versions.

The standard version puts the current system date into one integer array: day, month, and year.

The subroutine is called by:

| `call idate(` ***iarray*** `)`***Standard Version*** | | | |
|---|---|---|---|
| *iarray* | `INTEGER*4` | Output | `array`(3). Note the order: day, month, year. |

Example: `idate` (standard version):

```
demo% cat tidate.f
    INTEGER*4 iarray(3)
    call idate( iarray )
    write(*, "(" The date is: ",3i5)" )  iarray
    end
demo% f77 -silent tidate.f
demo% a.out
 The date is: 10 8 1998
demo%
```

The VMS `idate()` subroutine is called by:

| `call idate(` ***m***, ***d***, ***y*** `)`***VMS Version*** | | | |
|---|---|---|---|
| *m* | `INTEGER*4` | Output | Month (1 - 12) |
| *d* | `INTEGER*4` | Output | Day (1 - 7) |
| *y* | `INTEGER*4` | Output | Year (1 - 99) *Not year 2000 safe!* |

Using the VMS `idate()` routine will cause a warning message at link time and the first time the routine is called in execution.

**Note -** The VMS version of the `idate()` routine is not “Year 2000 Safe” because it returns only a two-digit value for the year. Programs that compute differences between dates using the output of this routine may not work properly after 31 December, 1999. Programs using this `idate()` routine will see a runtime warning message the first time the routine is called to alert the user. See `date_and_time()` as a possible alternate.

Example: `idate` (VMS version):

```
demo% cat titime.f
    INTEGER*4 m, d, y
    call idate ( m, d, y )
    write (*, "(" The date is: ",3i5)" ) m, d, y
    end
demo% f77 -silent tidateV.f -lV77
"titime.f", line 2: Warning: Subroutine "idate" is not safe after
                     year 2000; use "date_and_time" instead
demo% a.out
Computing time differences using the 2 digit year from subroutine
                     idate is not safe after year 2000.
 The date is:     7   10   98
```

## `ieee_flags,ieee_handler, sigfpe`: IEEE Arithmetic

These subprograms provide modes and status required to fully exploit ANSI/IEEE Std 754-1985 arithmetic in a FORTRAN program. They correspond closely to the functions *ieee_flags*(3M), *ieee_handler*(3M), and *sigfpe*(3).

Here is a summary:

**TABLE 1–2** IEEE Arithmetic Support Routines

| `ieeer = ieee_flags(` *action*`,`*mode*`,`*in*`,`*out* `)`<br>`ieeer = ieee_handler(`*action*`,`*exception*`,`*hdl* `)`<br>`ieeer = sigfpe(` *code*`,` *hdl* `)` | | |
|---|---|---|
| *action* | `character` | Input |
| *code* | `sigfpe_code_type` | Input |

**TABLE 1–2** IEEE Arithmetic Support Routines *(continued)*

| | | |
|---|---|---|
| *mode* | `character` | Input |
| *in* | `character` | Input |
| *exception* | `character` | Input |
| *hdl* | `sigfpe_handler_type` | Input |
| *out* | `character` | Output |
| Return value | `INTEGER*4` | Output |

See the Sun *Numerical Computation Guide* for details on how these functions can be used strategically.

If you use `sigfpe`, you must do your own setting of the corresponding trap-enable-mask bits in the floating-point status register. The details are in the SPARC architecture manual. The `libm` function `ieee_handler` sets these trap-enable-mask bits for you.

The character keywords accepted for *mode* and *exception* depend on the value of *action*.

**TABLE 1–3** ieee_flags(*action,mode,in,out*) Parameters and Actions

| | | |
|---|---|---|
| *action* = "clearall" | *mode, in, out,* unused; returns 0 | |
| *action* = "clear" clear *mode, in* *out* is unused; returns 0 | *mode* = "direction" | |
| | *mode* = "precision" *(on x86 platforms only)* | |
| | *mode* = "exception" | *in* = "inexact" *or* "division" *or* "underflow" *or* "overflow" *or* "invalid" *or* "all" *or* "common" |
| *action* = "set" set floating-point *mode,in* *out* is unused; returns 0 | *mode* = "direction" | *in* = "nearest' *or* "tozero' *or* "positive' *or* "negative" |
| | *mode* = "precision" *(on x86 only)* | *in* = "extended" *or* "double" *or* "single" |
| | *mode* = "exception" | *in* = "inexact" *or* "division" *or* "underflow" *or* "overflow" *or* "invalid" *or* "all" *or* "common" |

**TABLE 1–3** ieee_flags(*action*,*mode*,*in*,*out*) Parameters and Actions *(continued)*

| | | |
|---|---|---|
| *action* = "get"<br>test *mode* settings<br>*in, out* may be blank or one of the settings to test returns the current setting depending on *mode*, or "`not available`" The function returns 0 or the current exception flags if *mode* = "`exception`" | *mode* = "direction" | *out* = "nearest' *or* "tozero' *or* "positive' *or* "negative" |
| | *mode* = "precision" *(on x86 only)* | *out* = "extended" *or* "double" *or* "single" |
| | *mode* = "exception" | *out* = "inexact" *or* "division" *or* "underflow" *or* "overflow" *or* "invalid" *or* "all" *or* "common" |

**TABLE 1–4** ieee_handler(*action*,*in*,*out*) Parameters

| | |
|---|---|
| *action* = "clear"clear user exception handing of *in*; *out* is unused | *in* = "inexact" *or* "division" *or* "underflow" *or* "overflow" *or* "invalid" *or* "all" *or* "common" |
| *action* = "set" set user exception handing of *in*; *out* is address of handler routine, or `SIGFPE_DEFAULT`, or `SIGFPE_ABORT`, or `SIGFPE_IGNORE` defined in `f77/f77_floating point.h` | *in* = "inexact" *or* "division" *or* "underflow" *or* "overflow" *or* "invalid" *or* "all" *or* "common" |

Example 1: Set rounding direction to round toward zero, unless the hardware does not support directed rounding modes:

```
INTEGER*4 ieeer
character*1 mode, out, in
ieeer = ieee_flags( "set", "direction", "tozero", out )
```

Example 2: Clear rounding direction to default (round toward nearest):

```
character*1 out, in
ieeer = ieee_flags("clear","direction", in, out )
```

Example 3: Clear all accrued exception-occurred bits:

```
character*18 out
ieeer = ieee_flags( "clear", "exception", "all", out )
```

Example 4: Detect overflow exception as follows:

```
    character*18 out
    ieeer = ieee_flags( "get", "exception", "overflow", out )
    if (out .eq. "overflow" ) stop "overflow"
```

The above code sets `out` to `overflow` and `ieeer` to 25 (this value is platform dependent). Similar coding detects exceptions, such as `invalid` or `inexact`.

Example 5: `hand1.f`, write and use a signal handler (*Solaris 2*):

```
    external hand
    real r / 14.2 /,  s / 0.0 /
    i = ieee_handler( "set", "division", hand )
    t = r/s
    end

    INTEGER*4 function hand ( sig, sip, uap )
    INTEGER*4 sig, address
    structure /fault/
        INTEGER*4 address
    end structure
    structure /siginfo/
        INTEGER*4 si_signo
        INTEGER*4 si_code
        INTEGER*4 si_errno
        record /fault/ fault
    end structure
    record /siginfo/ sip
    address = sip.fault.address
    write (*,10) address
 10    format("Exception at hex address ", z8 )
    end
```

See the *Numerical Computation Guide*. See also: *floatingpoint*(3), *signal*(3), *sigfpe*(3), *f77_floatingpoint*(3F), *ieee_flags*(3M), and *ieee_handler*(3M).

## `f77_floatingpoint.h`: FORTRAN IEEE Definitions

The header file `f77_floatingpoint.h` defines constants and types used to implement standard floating-point according to ANSI/IEEE Std 754-1985.

Include the file in a FORTRAN 77 source program as follows:

```
#include "f77_floatingpoint.h"
```

Use of this include file requires preprocessing prior to FORTRAN compilation.The source file referencing this include file will automatically be preprocessed if the name has a `.F` or `.F90` extension.

Fortran 90 programs should include the file `f90/floatingpoint.h` instead.

IEEE Rounding Mode:

| | |
|---|---|
| `fp_direction_type` | The type of the IEEE rounding direction mode. The order of enumeration varies according to hardware. |

SIGFPE Handling:

| | |
|---|---|
| `sigfpe_code_type` | The type of a `SIGFPE` code. |
| `sigfpe_handler_type` | The type of a user-definable `SIGFPE` exception handler called to handle a particular `SIGFPE` code. |
| `SIGFPE_DEFAULT` | A macro indicating default `SIGFPE` exception handling: IEEE exceptions to continue with a default result and to abort for other `SIGFPE` codes. |
| `SIGFPE_IGNORE` | A macro indicating an alternate `SIGFPE` exception handling, namely to ignore and continue execution. |
| `SIGFPE_ABORT` | A macro indicating an alternate `SIGFPE` exception handling, namely to abort with a core dump. |

IEEE Exception Handling:

| | |
|---|---|
| `N_IEEE_EXCEPTION` | The number of distinct IEEE floating-point exceptions. |
| `fp_exception_type` | The type of the `N_IEEE_EXCEPTION` exceptions. Each exception is given a bit number. |
| `fp_exception_field_type` | The type intended to hold at least `N_IEEE_EXCEPTION` bits corresponding to the IEEE exceptions numbered by `fp_exception_type`. Thus, `fp_inexact` corresponds to the least significant bit and `fp_invalid` to the fifth least significant bit. Some operations can set more than one exception. |

IEEE Classification:

| | |
|---|---|
| `fp_class_type` | A list of the classes of IEEE floating-point values and symbols. |

Refer to the *Numerical Computation Guide*. See also *ieee_environment*(3M) and *f77_ieee_environment*(3F).

## `index,rindex,lnblnk`: Index or Length of Substring

These functions search through a character string:

| | |
|---|---|
| `index(`*a1*`,`*a2*`)` | Index of first occurrence of string *a2* in string *a1* |
| `rindex(`*a1*`,`*a2*`)` | Index of last occurrence of string *a2* in string *a1* |
| `lnblnk(`*a1*`)` | Index of last nonblank in string *a1* |

`index` has the following forms:

### `index`: First Occurrence of a Substring in a String

The index is an intrinsic function called by:

| ***n*** = `index(` ***a1***`,` ***a2*** `)` | | | |
|---|---|---|---|
| *a1* | `character` | Input | Main string |
| *a2* | `character` | Input | Substring |
| Return value | `INTEGER` | Output | *n*>0: Index of first occurrence of *a2* in *a1*<br>*n*=0: *a2* does not occur in *a1*. |

If declared `INTEGER*8`, `index()` will return an `INTEGER*8` value when compiled for a 64-bit environment and character variable *a1* is a very large character string (greater than 2 Gigabytes).

## `rindex`: Last Occurrence of a Substring in a String

The function is called by:

| `INTEGER*4 rindex`<br><br>***n*** `= rindex(` ***a1***`,` ***a2*** `)` | | | |
|---|---|---|---|
| *a1* | `character` | Input | Main string |
| *a2* | `character` | Input | Substring |
| Return value | `INTEGER*4` *or*`INTEGER*8` | Output | *n*>0: Index of last occurrence of *a2* in *a1*<br><br>*n*=0: *a2* does not occur in *a1*`INTEGER*8` returned in 64-bit environments |

## `lnblnk`: Last Nonblank in a String

The function is called by:

| ***n*** `= lnblnk(` ***a1*** `)` | | | |
|---|---|---|---|
| *a1* | `character` | Input | String |
| Return value | `INTEGER*4` *or*`INTEGER*8` | Output | *n*>0: Index of last nonblank in *a1*<br><br>*n*=0: *a1* is all nonblank `INTEGER*8` returned in 64-bit environments |

Example: `index()`, `rindex()`, `lnblnk()`:

```
*                      123456789012345678901
    character s*24 / "abcPDQxyz...abcPDQxyz" /
    INTEGER*4 declen, index, first, last, len, lnblnk, rindex
    declen = len( s )
    first = index( s, "abc" )
    last = rindex( s, "abc" )
    lastnb = lnblnk( s )
    write(*,*) declen, lastnb
    write(*,*) first, last
    end
demo% f77 -silent tindex.f
demo% a.out
24 21     <- declen is 24  because intrinsic len() returns the declared length of  s
1 13
```

**Note -** Programs compiled to run in a 64-bit environment must declare `index, rindex` and `lnblnk` (and their receiving variables) `INTEGER*8` to handle very large character strings.

## `inmax`: Return Maximum Positive Integer

The function is called by:

| *m* = `inmax()` | | | |
|---|---|---|---|
| Return value | `INTEGER*4` | Output | The maximum positive integer |

Example: `inmax`:

```
    INTEGER*4 inmax, m
    m = inmax()
    write(*,*) m
    end
demo% f77 -silent tinmax.f
demo% a.out
   2147483647
demo%
```

See also *libm_single*(3F) and *libm_double*(3F). See also the intrinsic function `ephuge()` described in the *FORTRAN 77 Language Reference Manual.*

## `ioinit`: Initialize I/O Properties

The `IOINIT` routine (FORTRAN 77 only) establishes properties of file I/O for files opened after the call to `IOINIT`. The file I/O properties that `IOINIT` controls are as follows:

- Carriage control: Recognize carriage control on any logical unit.
- Blanks/zeroes: Treat blanks in input data fields as blanks or zeroes.
- File position: Open files at beginning or at end-of-file.
- Prefix: Find and open files named *prefixNN*, 0 £ *NN* £ 19.

`IOINIT` does the following:

- Initializes global parameters specifying `f77` file I/O properties
- Opens logical units 0 through 19 with the specified file I/O properties—attaches externally defined files to logical units at runtime

### Persistence of File I/O Properties

The file I/O properties apply as long as the connection exists. If you close the unit, the properties no longer apply. The exception is the preassigned units 5 and 6, to which carriage control and blanks/zeroes apply at any time.

### Internal Flags

`IOINIT` uses labeled common to communicate with the runtime I/O system. It stores internal flags in the equivalent of the following labeled common block:

```
INTEGER*2 IEOF, ICTL, IBZR
COMMON /__IOIFLG/ IEOF, ICTL, IBZR ! Not in user name space
```

In releases prior to SC 3.0.1, the labeled common block was named `IOIFLG`. The name changed subsequently to `_ _IOIFLG` to prevent conflicts with any user-defined common blocks.

### Source Code

Some user needs are not satisfied with a generic version of `IOINIT`, so we provide the source code. It is written in FORTRAN 77. The location is:

*<install>*`/SUNWspro/SC5.0/src/ioinit.f`

where *<install>* is usually `/opt` for a standard installation of the Sun Fortran software package.

## Usage: `ioinit`

The `ioinit` subroutine is called by:

| `call ioinit (` ***cctl***`,` ***bzro***`,` ***apnd***`,` ***prefix***`,` ***vrbose*** `)` | | | |
|---|---|---|---|
| *cctl* | `logical` | Input | True: Recognize carriage control, all formatted output (except unit 0) |
| *bzro* | `logical` | Input | True: Treat trailing and imbedded blanks as zeroes. |
| *apnd* | `logical` | Input | True: Open files at EoF. Append. |
| *prefix* | `character*`*n* | Input | Nonblank: For unit *NN*, seek and open file *prefixNN* |
| *vrbose* | `logical` | Input | True: Report `ioinit` activity as it happens |

See also *getarg*(3F) and *getenv*(3F).

## Restrictions

Note the following restrictions:

- *prefix* can be no longer than 30 characters.
- A path name associated with an environment name can be no longer than 255 characters.

## Description of Arguments

These are the arguments for `ioinit`.

## *cctl* (Carriage Control)

By default, carriage control is not recognized on any logical unit. If *cctl* is `.TRUE.`, then carriage control is recognized on formatted output to all logical units, except unit 0, the diagnostic channel. Otherwise, the default is restored.

## *bzro* (Blanks)

By default, trailing and embedded blanks in input data fields are ignored. If *bzro* is `.TRUE.`, then such blanks are treated as zeros. Otherwise, the default is restored.

## *apnd* (Append)

By default, all files opened for sequential access are positioned at their beginning. It is sometimes necessary or convenient to open at the end-of-file, so that a write will append to the existing data. If *apnd* is `.TRUE.`, then files opened subsequently on any logical unit are positioned at their end upon opening. A value of `.FALSE.` restores the default behavior.

## *prefix* (Automatic File Connection)

If the argument *prefix* is a nonblank string, then names of the form *prefixNN* are sought in the program environment. The value associated with each such name found is used to open the logical unit *NN* for formatted sequential access.

This search and connection is provided only for *NN* between 0 and 19, inclusive. For *NN* > 19, nothing is done; see “Source Code” on page 44.

## *vrbose* (`IOINIT` Activity)

If the argument *vrbose* is `.TRUE.`, then `IOINIT` reports on its own activity.

Example: The program `myprogram` has the following `ioinit` call:

```
    call ioinit( .true., .false., .false., "FORT", .false.)
```

You can assign file name in at least two ways.

In `sh`:

```
demo$ FORT01=mydata
demo$ FORT12=myresults
demo$ export FORT01 FORT12
demo$ myprogram
```

In `csh`:

```
demo% setenv FORT01 mydata
demo% setenv FORT12 myresults
demo% myprogram
```

With either shell, the `ioinit` call in the above example gives these results:

- Open logical unit 1 to the file, `mydata`.
- Open logical unit 12 to the file, `myresults`.
- Both files are positioned at their beginning.
- Any formatted output has column 1 removed and interpreted as carriage control.
- Embedded and trailing blanks are to be ignored on input.

Example: `ioinit()`—list and compile:

```
demo% cat tioinit.f
    character*3  s
    call ioinit( .true., .false., .false., "FORT", .false.)
    do i = 1, 2
        read( 1, "(a3,i4)")  s, n
        write( 12, 10 ) s, n
    end do
10    format(a3,i4)
    end
demo% cat tioinit.data
abc 123
PDQ 789
demo% f77 -silent tioinit.f
demo%
```

You can set environment variables as follows, using either `sh` or `csh`:

`ioinit()`—`sh`:

```
demo$ FORT01=tioinit.data
demo$ FORT12=tioinit.au
demo$ export FORT01 FORT12
demo$
```

`ioinit()`—`csh`:

```
demo% a.out
demo% cat tioinit.au
abc 123
PDQ 789
```

`ioinit()`—Run and test:

```
demo% a.out
demo% cat tioinit.au
abc 123
PDQ 789
```

## itime: Current Time

itime puts the current system time into an integer array: hour, minute, and second. The subroutine is called by:

| call itime( ***iarray*** ) | | | |
|---|---|---|---|
| *iarray* | INTEGER*4 | Output | 3-element array:<br>*iarray*(1) = hour<br>*iarray*(2) = minute<br>*iarray*(3) = second |

Example: itime:

```
demo% cat titime.f
    INTEGER*4 iarray(3)
    call itime( iarray )
    write (*, "(" The time is: ",3i5)" )  iarray
    end
demo% f77 -silent titime.f
demo% a.out
 The time is: 15 42 35
```

See also *time*(3F), *ctime*(3F), and *fdate*(3F).

## kill: Send a Signal to a Process

The function is called by:

| ***status*** = `kill(` ***pid***, ***signum*** `)` | | | |
|---|---|---|---|
| *pid* | `INTEGER*4` | Input | Process ID of one of the user's processes |
| *signum* | `INTEGER*4` | Input | Valid signal number. See *signal*(3). |
| Return value | `INTEGER*4` | Output | *status*=0: OK<br>*status*>0: Error code |

Example (fragment): Send a message using `kill()`:

```
    INTEGER*4 kill, pid, signum
*     …
    status = kill( pid, signum )
    if ( status .ne. 0 ) stop "kill: error"
    write(*,*) "Sent signal ", signum, " to process ", pid
    end
```

The function sends signal *signum*, and integer signal number, to the process *pid*. Valid signal numbers are listed in the C include file `/usr/include/sys/signal.h`

See also: *kill*(2), *signal*(3), *signal*(3F), *fork*(3F), and *perror*(3F).

## `libm` Math Functions

The following functions and subroutines are part of the math library libm. Some routines are intrinsics and return the same data type (single precision, double precision, or quad precision) as their argument. The rest are non-intrinsics that take a specific data type as an argument and return the same. These non-intrinsics do have to be declared in the routine referencing them.

### `libm` Intrinsic Functions

Here is a list of the intrinsic functions in `libm`. You need not put them in a type statement. These functions take single, double, or quad precision data as arguments and return the same.

| | | |
|---|---|---|
| sqrt(x) | asin(x) | cosd(x) |
| log(x) | acos(x) | asind(x) |
| log10(x) | atan(x) | acosd(x) |
| exp(x) | atan2(x,y) | atand(x) |
| x**y | sinh(x) | atan2d(x,y) |
| sin(x) | cosh(x) | aint(x) |
| cos(x) | tanh(x) | anint(x) |
| tan(x) | sind(x) | nint(x) |

The
functions `sind(x), cosd(x), asind(x), acosd(x), atand(x), atan2d(x,y)` are not considered intrinsics by the FORTRAN 77 standard.

## `libm_double`: Double-Precision Functions

The following subprograms are double-precision `libm` functions and subroutines.

In general, these functions do *not* correspond to standard FORTRAN generic intrinsic functions—data types are determined by the usual data typing rules.

Example: Subroutine and non-Intrinsic double-precision functions:

```
DOUBLE PRECISION c, d_acosh, d_hypot, d_infinity, s, x, y, z
...
z = d_acosh( x )
i = id_finite( x )
z = d_hypot( x, y )
z = d_infinity()
CALL d_sincos( x, s, c )
```

These `DOUBLE PRECISION` functions need to appear in a `DOUBLE PRECISION` statement.

Refer to the C library man pages for details: the man page for `d_acos(x)` is *acos*(3M)

**TABLE 1–5** Double Precision `libm` Functions

| | | | |
|---|---|---|---|
| d_acos( x ) | DOUBLE PRECISION | Function | arc cosine |
| d_acosd( x ) | DOUBLE PRECISION | Function | – |
| d_acosh( x ) | DOUBLE PRECISION | Function | arc cosh |
| d_acosp( *x* ) | DOUBLE PRECISION | Function | – |
| d_acospi( x ) | DOUBLE PRECISION | Function | – |
| d_atan( x ) | DOUBLE PRECISION | Function | arc tangent |
| d_atand( x ) | DOUBLE PRECISION | Function | – |
| d_atanh( x ) | DOUBLE PRECISION | Function | arc tanh |
| d_atanp( x ) | DOUBLE PRECISION | Function | – |
| d_atanpi( x ) | DOUBLE PRECISION | Function | – |
| d_asin( x ) | DOUBLE PRECISION | Function | arc sine |
| d_asind( x ) | DOUBLE PRECISION | Function | – |
| d_asinh( x ) | DOUBLE PRECISION | Function | arc sinh |
| d_asinp( x ) | DOUBLE PRECISION | Function | – |
| d_asinpi( x ) | DOUBLE PRECISION | Function | – |

**TABLE 1–5** Double Precision `libm` Functions *(continued)*

| | | | |
|---|---|---|---|
| d_atan2(( y, x ) | DOUBLE PRECISION | Function | arc tangent |
| d_atan2d( y, x ) | DOUBLE PRECISION | Function | – |
| d_atan2pi( y, x ) | DOUBLE PRECISION | Function | – |
| d_cbrt( *x* ) | DOUBLE PRECISION | Function | cube root |
| d_ceil( *x* ) | DOUBLE PRECISION | Function | ceiling |
| d_copysign( *x*, *x* ) | DOUBLE PRECISION | Function | – |
| d_cos( x ) | DOUBLE PRECISION | Function | cosine |
| d_cosd( x ) | DOUBLE PRECISION | Function | – |
| d_cosh( x ) | DOUBLE PRECISION | Function | hyperb cos |
| d_cosp( x ) | DOUBLE PRECISION | Function | – |
| d_cospi( x ) | DOUBLE PRECISION | Function | – |
| d_erf( x ) | DOUBLE PRECISION | Function | error func |
| d_erfc( x ) | DOUBLE PRECISION | Function | – |

**TABLE 1–5** Double Precision `libm` Functions *(continued)*

| | | | |
|---|---|---|---|
| d_expm1( x ) | DOUBLE PRECISION | Function | (e**x)-1 |
| d_floor( x ) | DOUBLE PRECISION | Function | floor |
| d_hypot( x, y ) | DOUBLE PRECISION | Function | hypotenuse |
| d_infinity( ) | DOUBLE PRECISION | Function | – |
| d_j0( x ) | DOUBLE PRECISION | Function | Bessel |
| d_j1( x ) | DOUBLE PRECISION | Function | – |
| d_jn( x ) | DOUBLE PRECISION | Function | – |
| id_finite( x ) | INTEGER | Function | |
| id_fp_class( x ) | INTEGER | Function | |
| id_ilogb( x ) | INTEGER | Function | |
| id_irint( x ) | INTEGER | Function | |
| id_isinf( x ) | INTEGER | Function | |
| id_isnan( x ) | INTEGER | Function | |
| id_isnormal( x ) | INTEGER | Function | |
| id_issubnormal( x ) | INTEGER | Function | |
| id_iszero( x ) | INTEGER | Function | |
| id_signbit( x ) | INTEGER | Function | |

**TABLE 1–5** Double Precision `libm` Functions *(continued)*

| | | | |
|---|---|---|---|
| `d_addran()` | DOUBLE PRECISION | Function | random number generators |
| `d_addrans(x, p, l, u)` | n/a | Subroutine | |
| `d_lcran()` | DOUBLE PRECISION | Function | |
| `d_lcrans(x, p, l, u )` | n/a | Subroutine | |
| `d_shufrans(x, p, l,u)` | n/a | Subroutine | |
| `d_lgamma( x )` | DOUBLE PRECISION | Function | log gamma |
| `d_logb( x )` | DOUBLE PRECISION | Function | – |
| `d_log1p( x )` | DOUBLE PRECISION | Function | – |
| `d_log2( x )` | DOUBLE PRECISION | Function | – |

**TABLE 1–5** Double Precision `libm` Functions *(continued)*

| | | | |
|---|---|---|---|
| `d_max_normal()` | `DOUBLE PRECISION` | `Function` | |
| `d_max_subnormal()` | `DOUBLE PRECISION` | `Function` | |
| `d_min_normal()` | `DOUBLE PRECISION` | `Function` | |
| `d_min_subnormal()` | `DOUBLE PRECISION` | `Function` | |
| `d_nextafter( x, y )` | `DOUBLE PRECISION` | `Function` | |
| `d_quiet_nan( n )` | `DOUBLE PRECISION` | `Function` | |
| `d_remainder( x, y )` | `DOUBLE PRECISION` | `Function` | |
| `d_rint( x )` | `DOUBLE PRECISION` | `Function` | |
| `d_scalb( x, y )` | `DOUBLE PRECISION` | `Function` | |
| `d_scalbn( x, n )` | `DOUBLE PRECISION` | `Function` | |
| `d_signaling_nan( n )` | `DOUBLE PRECISION` | `Function` | |
| `d_significand( x )` | `DOUBLE PRECISION` | `Function` | |
| `d_sin( x )` | `DOUBLE PRECISION` | `Function` | `sine` |
| `d_sind( x )` | `DOUBLE PRECISION` | `Function` | – |
| `d_sinh( x )` | `DOUBLE PRECISION` | `Function` | `hyperb sine` |
| `d_sinp( x )` | `DOUBLE PRECISION` | `Function` | – |
| `d_sinpi( x )` | `DOUBLE PRECISION` | `Function` | – |

**TABLE 1–5** Double Precision `libm` Functions *(continued)*

| | | | |
|---|---|---|---|
| `d_sincos( x, s, c )` | n/a | `Subroutine` | `sine and cosine` |
| `d_sincosd( x, s, c )` | n/a | `Subroutine` | – |
| `d_sincosp( x, s, c )` | n/a | `Subroutine` | – |
| `d_sincospi( x, s, c )` | n/a | `Subroutine` | |
| `d_tan( x )` | `DOUBLE PRECISION` | `Function` | `tangent` |
| `d_tand( x )` | `DOUBLE PRECISION` | `Function` | – |
| `d_tanh( x )` | `DOUBLE PRECISION` | `Function` | `hyperb tan` |
| `d_tanp( x )` | `DOUBLE PRECISION` | `Function` | – |
| `d_tanpi( x )` | `DOUBLE PRECISION` | `Function` | – |
| `d_y0( x )` | `DOUBLE PRECISION` | `Function` | `bessel` |
| `d_y1( x )` | `DOUBLE PRECISION` | `Function` | – |
| `d_yn( n, x )` | `DOUBLE PRECISION` | `Function` | – |

- Variables `c`, `l`, `p`, `s`, `u`, `x`, and `y` are of type `DOUBLE PRECISION`.
- Explicitly type these functions on a `DOUBLE PRECISION` statement or with an appropriate `IMPLICIT` statement).
- `sind(x)`, `asind(x)`, … take *degrees* rather than *radians*.

See also: *intro*(3M) and the *Numerical Computation Guide*.

## `libm_quadruple`: Quad-Precision Functions

These subprograms are `quadruple`-precision (`REAL*16`) `libm` functions and subroutines *(SPARC only)*.

In general, these do *not* correspond to standard generic intrinsic functions; data types are determined by the usual data typing rules.

Samples: Quadruple precision functions:

```
REAL*16 c, q_acosh, q_hypot, q_infinity, s, x, y, z
...
z = q_acosh( x )
i = iq_finite( x )
z = q_hypot( x, y )
z = q_infinity()
CALL q_sincos( x, s, c )
```

The quadruple precision functions must appear in a `REAL*16` statement

**TABLE 1–6** Quadruple-Precision `libm` Functions

| | | |
|---|---|---|
| `q_copysign( x, y )` | `REAL*16` | `Function` |
| `q_fabs( x )` | `REAL*16` | `Function` |
| `q_fmod( x )` | `REAL*16` | `Function` |
| `q_infinity( )` | `REAL*16` | `Function` |
| `iq_finite( x )` | `INTEGER` | `Function` |
| `iq_fp_class( x )` | `INTEGER` | `Function` |
| `iq_ilogb( x )` | `INTEGER` | `Function` |
| `iq_isinf( x )` | `INTEGER` | `Function` |
| `iq_isnan( x )` | `INTEGER` | `Function` |
| `iq_isnormal( x )` | `INTEGER` | `Function` |
| `iq_issubnormal( x )` | `INTEGER` | `Function` |
| `iq_iszero( x )` | `INTEGER` | `Function` |
| `iq_signbit( x )` | `INTEGER` | `Function` |
| `q_max_normal()` | `REAL*16` | `Function` |
| `q_max_subnormal()` | `REAL*16` | `Function` |
| `q_min_normal()` | `REAL*16` | `Function` |
| `q_min_subnormal()` | `REAL*16` | `Function` |
| `q_nextafter( x, y )` | `REAL*16` | `Function` |
| `q_quiet_nan( n )` | `REAL*16` | `Function` |
| `q_remainder( x, y )` | `REAL*16` | `Function` |
| `q_scalbn( x, n )` | `REAL*16` | `Function` |
| `q_signaling_nan( n )` | `REAL*16` | `Function` |

- The variables `c`, `l`, `p`, `s`, `u`, `x`, and `y` are of type quadruple precision.
- Explicitly type these functions with a `REAL*16` statement or with an appropriate `IMPLICIT` statement.

- `sind(x)`, `asind(x)`, … take *degrees* rather than *radians*.

If you need to use any other quadruple-precision `libm` function, you can call it using `$PRAGMA C(`*fcn*`)` before the call. For details, see the chapter on the C–FORTRAN interface in the *Fortran Programming Guide*.

## `libm_single`: Single-Precision Functions

These subprograms are single-precision `libm` functions and subroutines.

In general, the functions below provide access to single-precision `libm` functions that do *not* correspond to standard FORTRAN generic intrinsic functions—data types are determined by the usual data typing rules.

Samples: Single-precision `libm` functions:

```
REAL c, s, x, y, z
..
z = r_acosh( x )
i = ir_finite( x )
z = r_hypot( x, y )
z = r_infinity()
CALL r_sincos( x, s, c )
```

These functions need not be explicitly typed with a `REAL` statement as long as default typing holds. (Variables beginning with “`r`” are `REAL`, with “`i`” are `INTEGER`.)

For details on these routines, see the C math library man pages (3M). For example, for `r_acos(x)` see the *acos*(3M) man page.

**TABLE 1–7** Single-Precision `libm` functions

| | | | |
|---|---|---|---|
| r_acos( x ) | REAL | Function | arc cosine |
| r_acosd( x ) | REAL | Function | – |
| r_acosh( x ) | REAL | Function | arc cosh |
| r_acosp( x ) | REAL | Function | – |
| r_acospi( x ) | REAL | Function | – |
| r_atan( x ) | REAL | Function | arc tangent |
| r_atand( x ) | REAL | Function | – |
| r_atanh( x ) | REAL | Function | arc tanh |
| r_atanp( x ) | REAL | Function | – |
| r_atanpi( x ) | REAL | Function | – |
| r_asin( x ) | REAL | Function | arc sine |
| r_asind( x ) | REAL | Function | – |
| r_asinh( x ) | REAL | Function | arc sinh |
| r_asinp( x ) | REAL | Function | – |
| r_asinpi( x ) | REAL | Function | – |
| r_atan2(( y, x ) | REAL | Function | arc tangent |
| r_atan2d( y, x ) | REAL | Function | – |
| r_atan2pi( y, x ) | REAL | Function | – |
| r_cbrt( x ) | REAL | Function | cube root |
| r_ceil( x ) | REAL | Function | ceiling |
| r_copysign( x, y ) | REAL | Function | – |

**TABLE 1–7** Single-Precision `libm` functions *(continued)*

| | | | |
|---|---|---|---|
| `r_cos( x )` | `REAL` | `Function` | `cosine` |
| `r_cosd( x )` | `REAL` | `Function` | – |
| `r_cosh( x )` | `REAL` | `Function` | `hyperb cos` |
| `r_cosp( x )` | `REAL` | `Function` | – |
| `r_cospi( x )` | `REAL` | `Function` | – |
| `r_erf( x )` | `REAL` | `Function` | `err function` |
| `r_erfc( x )` | `REAL` | `Function` | – |
| `r_expm1( x )` | `REAL` | `Function` | `(e**x)-1` |
| `r_floor( x )` | `REAL` | `Function` | `floor` |
| `r_hypot( x, y )` | `REAL` | `Function` | `hypotenuse` |
| `r_infinity( )` | `REAL` | `Function` | `bessel` |
| `r_j0( x )` | `REAL` | `Function` | – |
| `r_j1( x )` | `REAL` | `Function` | – |
| `r_jn( x )` | `REAL` | `Function` | – |
| `ir_finite( x )` | `INTEGER` | `Function` | – |
| `ir_fp_class( x )` | `INTEGER` | `Function` | – |
| `ir_ilogb( x )` | `INTEGER` | `Function` | – |
| `ir_irint( x )` | `INTEGER` | `Function` | – |
| `ir_isinf( x )` | `INTEGER` | `Function` | – |
| `ir_isnan( x )` | `INTEGER` | `Function` | – |
| `ir_isnormal( x )` | `INTEGER` | `Function` | – |
| `ir_issubnormal( x )` | `INTEGER` | `Function` | – |
| `ir_iszero( x )` | `INTEGER` | `Function` | – |
| `ir_signbit( x )` | `INTEGER` | `Function` | – |

**TABLE 1–7** Single-Precision `libm` functions *(continued)*

| | | | |
|---|---|---|---|
| `r_addran()` | `REAl` | `Function` | `random number` |
| `r_addrans( x, p, l, u )` | n/a | `Subroutine` | – |
| `r_lcran()` | `REAL` | `Function` | – |
| `r_lcrans( x, p, l, u )` | n/a | `Subroutine` | – |
| `r_shufrans(x, p, l, u)` | n/a | `Subroutine` | – |
| `r_lgamma( x )` | `REAL` | `Function` | `log` gamma |
| `r_logb( x )` | `REAL` | `Function` | – |
| `r_log1p( x )` | `REAL` | `Function` | – |
| `r_log2( x )` | `REAL` | `Function` | – |
| `r_max_normal()` | `REAL` | `Function` | |
| `r_max_subnormal()` | `REAL` | `Function` | |
| `r_min_normal()` | `REAL` | `Function` | |
| `r_min_subnormal()` | `REAL` | `Function` | |
| `r_nextafter( x, y )` | `REAL` | `Function` | |
| `r_quiet_nan( n )` | `REAL` | `Function` | |
| `r_remainder( x, y )` | `REAL` | `Function` | |
| `r_rint( x )` | `REAL` | `Function` | |
| `r_scalb( x, y )` | `REAL` | `Function` | |
| `r_scalbn( x, n )` | `REAL` | `Function` | |
| `r_signaling_nan( n )` | `REAL` | `Function` | |
| `r_significand( x )` | `REAL` | `Function` | |

**TABLE 1–7** Single-Precision `libm` functions *(continued)*

| | | | |
|---|---|---|---|
| `r_sin( x )` | REAL | Function | sine |
| `r_sind( x )` | REAL | Function | – |
| `r_sinh( x )` | REAL | Function | hyperb sin |
| `r_sinp( x )` | REAL | Function | – |
| `r_sinpi( x )` | REAL | Function | – |
| `r_sincos( x, s, c )` | n/a | Subroutine | sine & cosine |
| `r_sincosd( x, s, c )` | n/a | Subroutine | – |
| `r_sincosp( x, s, c )` | n/a | Subroutine | – |
| `r_sincospi( x, s, c )` | n/a | Subroutine | – |
| `r_tan( x )` | REAL | Function | tangent |
| `r_tand( x )` | REAL | Function | – |
| `r_tanh( x )` | REAL | Function | hyperb tan |
| `r_tanp( x )` | REAL | Function | – |
| `r_tanpi( x )` | REAL | Function | – |
| `r_y0( x )` | REAL | Function | bessel |
| `r_y1( x )` | REAL | Function | – |
| `r_yn( n, x )` | REAL | Function | – |

- Variables `c`, `l`, `p`, `s`, `u`, `x`, and `y` are of type `REAL`.
- Type these functions as explicitly REAL if an IMPLICIT statement is in effect that types names starting with “`r`” to some other date type.
- `sind(x)`, `asind(x)`, … take *degrees* rather than *radians*.

See also: *intro*(3M) and the *Numerical Computation Guide*.

## `link`, `symlnk`: Make a Link to an Existing File

`link` creates a link to an existing file. `symlink` creates a symbolic link to an existing file.

The functions are called by:

| | | | |
|---|---|---|---|
| ***status*** = `link(` ***name1***, ***name2*** `)` | | | |
| `INTEGER*4 symlnk`<br><br>*status* = `symlnk(` *name1*, *name2* `)` | | | |
| *name1* | `character*`*n* | Input | Path name of an existing file |
| *name2* | `character*`*n* | Input | Path name to be linked to the file, *name1*.<br><br>*name2* must not already exist. |
| Return value | `INTEGER*4` | Output | *status*=0: OK<br><br>*status*>0: System error code |

## `link:` Create a Link to an Existing File

Example 1: `link`: Create a link named `data1` to the file, `tlink.db.data.1`:

```
demo% cat tlink.f
    character*34 name1/"tlink.db.data.1"/, name2/"data1"/
    integer*4 link, status
    status = link( name1, name2 )
    if ( status .ne. 0 ) stop "link: error"
    end
demo% f77 -silent tlink.f
demo% ls -l data1
data1 not found
demo% a.out
demo% ls -l data1
-rw-rw-r-- 2 generic 2 Aug 11 08:50 data1
demo%
```

## `symlnk`: Create a Symbolic Link to an Existing File

Example 2: `symlnk`: Create a symbolic link named `data1` to the file, `tlink.db.data.1`:

```
demo% cat tsymlnk.f
    character*34 name1/"tlink.db.data.1"/, name2/"data1"/
    INTEGER*4 status, symlnk
    status = symlnk( name1, name2 )
    if ( status .ne. 0 ) stop "symlnk: error"
    end
demo% f77 -silent tsymlnk.f
demo% ls -l data1
data1 not found
demo% a.out
demo% ls -l data1
lrwxrwxrwx 1 generic 15 Aug 11 11:09 data1 -> tlink.db.data.1
demo%
```

See also: *link*(2), *symlink*(2), *perror*(3F), and *unlink*(3F).

Note: the path names cannot be longer than `MAXPATHLEN` as defined in `<sys/param.h>`.

## `loc`: Return the Address of an Object

This intrinsic function is called by:

| ***k*** = `loc(` ***arg*** `)` | | | |
|---|---|---|---|
| *arg* | Any type | Input | Variable or array |
| Return value | `INTEGER*4` *-or-* `INTEGER*8` | Output | Address of *arg* |
| | Returns an `INTEGER*8` pointer when compiled to run in a 64-bit environment with `-xarch=v9`. See Note below. | | |

Example: `loc`:

```
INTEGER*4 k, loc
real arg / 9.0 /
k = loc( arg )
write(*,*) k
end
```

---

**Note -** Programs compiled to run in a 64-bit environment should declare `INTEGER*8` the variable receiving output from the `loc()` function.

---

## `long`, `short`: Integer Object Conversion

`long` and `short` handle integer object conversions between `INTEGER*4` and `INTEGER*2`, and is especially useful in subprogram call lists.

### `long`: Convert a Short Integer to a Long Integer

The function is called by:

| `call` ***ExpecLong***`( long(`***int2***`) )` | | |
|---|---|---|
| *int2* | `INTEGER*2` | Input |
| Return value | `INTEGER*4` | Output |

### `short`: Convert a Long Integer to a Short Integer

The function is:

| INTEGER*2 short<br><br>call ***ExpecShort***( short(***int4***) ) | | |
|---|---|---|
| *int4* | INTEGER*4 | Input |
| Return value | INTEGER*2 | Output |

Example (fragment): `long()` and `short()`:

```
integer*4 int4/8/, long
integer*2 int2/8/, short
call ExpecLong( long(int2) )
call ExpecShort( short(int4) )
…
end
```

*ExpecLong* is some subroutine called by the user program that expects a *long* (`INTEGER*4`) integer argument. Similarly, *ExpecShort* expects a *short* (`INTEGER*2`) integer argument.

`long` is useful if constants are used in calls to library routines and the code is compiled with the `-i2` option.

`short` is useful in similar context when an otherwise long object must be passed as a short integer. Passing an integer to short that is too large in magnitude does not cause an error, but will result in unexpected behavior.

## `longjmp`, `isetjmp`: Return to Location Set by `isetjmp`

`isetjmp` sets a location for `longjmp`; `longjmp` returns to that location.

### `isetjmp`: Set the Location for `longjmp`

This intrinsic function is called by:

| ***ival*** = `isetjmp(` ***env*** `)` | | | |
|---|---|---|---|
| *env* | `INTEGER*4` | Output | *env* is a 12-element integer array. In 64-bit environments it must be declared `INTEGER*8` |
| Return value | `INTEGER*4` | Output | *ival* = 0 if `isetjmp` is called explicitly<br><br>*ival* ¬= 0 if `isetjmp` is called through `longjmp` |

## `longjmp`: Return to the location set by `isetjmp`

The subroutine is called by:

| **call** `longjmp(` ***env***`,` ***ival*** `)` | | | |
|---|---|---|---|
| *env* | `INTEGER*4` | Input | *env* is the 12-word integer array initialized by `isetjmp`. In 64-bit environments it must be declared `INTEGER*8` |
| *ival* | `INTEGER*4` | Output | *ival* = 0 if `isetjmp` is called explicitly<br><br>*ival* ¬= 0 if `isetjmp` is called through `longjmp` |

## Description

The `isetjmp` and `longjmp` routines are used to deal with errors and interrupts encountered in a low-level routine of a program. They are `f77` intrinsics.

These routines should be used only as a last resort. They require discipline, and are not portable. Read the man page, *setjmp* (3V), for bugs and other details.

`isetjmp` saves the stack environment in *env*. It also saves the register environment.

`longjmp` restores the environment saved by the last call to `isetjmp`, and returns in such a way that execution continues as if the call to `isetjmp` had just returned the value *ival*.

The integer expression *ival* returned from `isetjmp` is zero if `longjmp` is not called, and nonzero if `longjmp` is called.

Example: Code fragment using `isetjmp` and `longjmp`:

```
INTEGER*4  env(12)
common /jmpblk/ env
j = isetjmp( env )
if ( j .eq. 0 ) then
call  sbrtnA
else
    call error_processor
end if
end
subroutine sbrtnA
INTEGER*4  env(12)
common /jmpblk/ env
call longjmp( env, ival )
return
end
```

## Restrictions

- You must invoke `isetjmp` before calling `longjmp`.
- The *env* integer array argument to `isetjmp` and longjmp must be at least 12 elements long.
- You must pass the *env* variable from the routine that calls `isetjmp` to the routine that calls `longjmp`, either by common or as an argument.
- `longjmp` attempts to clean up the stack. `longjmp` must be called from a lower call-level than `isetjmp`.
- Passing `isetjmp` as an argument that is a procedure name does not work.

See *setjmp*(3V).

# `malloc, malloc64`: Allocate Memory and Get Address

The `malloc()` function is called by:

| ***k*** = `malloc(` ***n*** `)` | | | |
|---|---|---|---|
| *n* | `INTEGER*4` | Input | Number of bytes of memory |
| Return value | `INTEGER*4` *or* `INTEGER*8` | Output | *k*>0: *k*=address of *the* start of the block of memory allocated<br>*k*=0: Error |
| | An `INTEGER*8` pointer value is returned when compiled for a 64-bit environment with `-xarch=v9`. See Note below. | | |

---

**Note -** Programs compiled to run on 64-bit environments such as Solaris 7 must declare the `malloc()` function and the variables receiving its output as `INTEGER*8`. Portability issues can be solved by using `malloc64()` instead of `malloc()` in programs that must run in both 32-bit or 64-bit environments.

---

The function `malloc64()` is provided to make programs portable between 32-bit and 64-bit environments:

| ***k*** = `malloc64(` ***n*** `)` | | | |
|---|---|---|---|
| *n* | `INTEGER*8` | Input | Number of bytes of memory |
| Return value | `INTEGER*8` | Output | *k*>0: *k*=address of *the* start of the block of memory allocated<br>*k*=0: Error |

These functions allocate an area of memory and return the address of the start of that area. (In a 64-bit environment, this returned byte address may be outside the `INTEGER*4` numerical range—the receiving variables must be declared `INTEGER*8` to avoid truncation of the memory address.) The region of memory is not initialized in any way, and it should not be assumed to be preset to anything, especially zero!

Example: Code fragment using `malloc()`:

```
parameter (NX=1000)
pointer ( p1, X )
real*4 X(NX)
…
p1 = malloc( NX*4 )
if ( p1 .eq. 0 ) stop "malloc: cannot allocate"
do 11 i=1,NX
```

```
11    X(i) = 0.
    …
    end
```

In the above example, we acquire 4,000 bytes of memory, pointed to by p1, and initialize it to zero.

See also “`free`: Deallocate Memory Allocated by Malloc ” on page 20.

## `mvbits`: Move a Bit Field

The subroutine is called by:

| `call mvbits(` *src*, *ini1*, *nbits*, *des*, *ini2* `)` | | | |
|---|---|---|---|
| *src* | `INTEGER*4` | Input | Source |
| *ini1* | `INTEGER*4` | Input | Initial bit position in the source |
| *nbits* | `INTEGER*4` | Input | Number of bits to move |
| *des* | `INTEGER*4` | Output | Destination |
| *ini2* | `INTEGER*4` | Input | Initial bit position in the destination |

Example: `mvbits`:

```
demo% cat mvb1.f
* mvb1.f -- From src, initial bit 0, move 3 bits to des, initial bit 3.
*    src    des
* 543210 543210 ¨ Bit numbers
* 000111 000001 ¨ Values before move
* 000111 111001 ¨ Values after move
    INTEGER*4 src, ini1, nbits, des, ini2
    data src, ini1, nbits, des, ini2
&            / 7,    0,     3,   1,    3 /
    call mvbits ( src, ini1, nbits, des, ini2 )
    write (*,"(5o3)") src, ini1, nbits, des, ini2
    end
demo% f77 -silent mvb1.f
demo% a.out
  7  0  3 71  3
demo%
```

Note the following:

- Bits are numbered 0 to 31, from least significant to most significant.
- `mvbits` changes only bits *ini2* through *ini2*+*nbits*`-1` of the *des* location, and no bits of the *src* location.
- The restrictions are:

  - *ini1* + *nbits* >= 32
  - *ini2* + *nbits* <= 32

## `perror`, `gerror`, `ierrno`: Get System Error Messages

These routines perform the following functions:

| | |
|---|---|
| `perror` | Print a message to FORTRAN logical unit 0, `stderr`. |
| `gerror` | Get a system error message (of the last detected system error) |
| `ierrno` | Get the error number of the last detected system error. |

### `perror`: Print Message to Logical Unit 0, `stderr`

The subroutine is called by:

| `call perror(` ***string*** `)` | | | |
|---|---|---|---|
| *string* | `character*`*n* | Input | The message. It is written preceding the standard error message for the last detected system error. |

Example 1:

```
call perror( "file is for formatted I/O" )
```

## `gerror`: Get Message for Last Detected System Error

The subroutine or function is called by:

| `call gerror( `***string***` )` | | | |
|---|---|---|---|
| *string* | `character*`*n* | Output | Message for the last detected system error |

Example 2: `gerror()` as a subroutine:

```
character string*30
…
call gerror ( string )
write(*,*) string
```

Example 3: `gerror`() as a function; *string* not used:

```
character gerror*30, z*30
…
z = gerror( )
write(*,*) z
```

## `ierrno`: Get Number for Last Detected System Error

The function is called by:

| ***n*** `= ierrno()` | | | |
|---|---|---|---|
| Return value | `INTEGER*4` | Output | Number of last detected system error |

This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

Example 4: `ierrno()`:

```
INTEGER*4 ierrno, n
…
n = ierrno()
write(*,*) n
```

See also *intro*(2) and *perror*(3).

Note:

- *string* in the call to `perror` cannot be longer than 127 characters.
- The length of the string returned by `gerror` is determined by the calling program.
- Runtime I/O error codes for f77 and f90 are listed in the *Fortran User's Guide*.

## `putc`, `fputc`: Write a Character to a Logical Unit

`putc` writes to logical unit 6, normally the control terminal output.

`fputc` writes to a logical unit.

These functions write a character to the file associated with a FORTRAN logical unit bypassing normal FORTRAN I/O.

Do not mix normal FORTRAN output with output by these functions on the same unit.

### `putc`: Write to Logical Unit 6

The function is called by:

| INTEGER*4 putc<br><br>***status*** = putc( ***char*** ) | | | |
|---|---|---|---|
| *char* | character | Input | The character to write to the unit |
| Return value | INTEGER*4 | Output | *status*=0: OK<br><br>*status*>0: System error code |

Example: putc():

```
    character char, s*10 / "OK by putc" /
    INTEGER*4 putc, status
    do i = 1, 10
        char = s(i:i)
        status = putc( char )
    end do
    status = putc( "\n" )
    end
demo% f77 -silent tputc.f
demo% a.out
OK by putc
demo%
```

## fputc: Write to Specified Logical Unit

The function is called by:

| INTEGER*4 fputc<br><br>***status*** = fputc( ***lunit*, *char*** ) | | | |
|---|---|---|---|
| *lunit* | INTEGER*4 | Input | The unit to write to |
| *char* | character | Input | The character to write to the unit |
| Return value | INTEGER*4 | Output | *status*=0: OK<br><br>*status*>0: System error code |

Example: `fputc()`:

```
    character char, s*11 / "OK by fputc" /
    INTEGER*4 fputc, status
    open( 1, file="tfputc.data")
    do i = 1, 11
        char = s(i:i)
        status = fputc( 1, char )
    end do
    status = fputc( 1, "\n" )
    end
demo% f77 -silent tfputc.f
demo% a.out
demo% cat tfputc.data
OK by fputc
demo%
```

See also *putc*(3S), *intro*(2), and *perror*(3F).

## `qsort,qsort64`: Sort the Elements of a One-dimensional Array

The subroutine is called by:

| call qsort( ***array***, ***len***, ***isize***, ***compar*** )<br>call qsort64( ***array***, ***len8***, ***isize8***, ***compar*** ) | | | |
|---|---|---|---|
| *array* | `array` | Input | Contains the elements to be sorted |
| *len* | `INTEGER*4` | Input | Number of elements in the array. |
| *len8* | `INTEGER*8` | Input | Number of elements in the array |
| *isize* | `INTEGER*4` | Input | Size of an element, typically:<br>4 for integer or real<br>8 for double precision or complex<br>16 for double complex<br>Length of character object for character arrays |

| call qsort( ***array***, ***len***, ***isize***, ***compar*** )<br>call qsort64( ***array***, ***len8***, ***isize8***, ***compar*** ) | | | |
|---|---|---|---|
| *isize8* | INTEGER*8 | Input | Size of an element, typically:<br>4_8 for integer or real<br>8_8 for double precision or complex<br>16_8 for double complex<br>Length of character object for character arrays |
| *compar* | function name | Input | Name of a user-supplied INTEGER*2 function.<br>Determines sorting order: compar(*arg1*,*arg2*) |

Use qsort64 in 64-bit environments with arrays larger than 2 Gbytes. Be sure to specify the array length, *len8*, and the element size, *isize8*, as INTEGER*8 data. Use the Fortran 90 style constants to explicitly specify INTEGER*8 constants.

The compar(*arg1*,*arg2*) arguments are elements of *array*, returning:

| | |
|---|---|
| Negative | If *arg1* is considered to precede *arg2* |
| Zero | If *arg1* is equivalent to *arg2* |
| Positive | If *arg1* is considered to follow *arg2* |

For example:

```
demo% cat tqsort.f
    external compar
    integer*2 compar
    INTEGER*4 array(10)/5,1,9,0,8,7,3,4,6,2/, len/10/, isize/4/
    call qsort( array, len, isize, compar )
    write(*,"(10i3)") array
    end
    integer*2 function compar( a, b )
    INTEGER*4 a, b
    if ( a .lt. b ) compar = -1
    if ( a .eq. b ) compar = 0
    if ( a .gt. b ) compar = 1
    return
    end
demo% f77 -silent tqsort.f
demo% a.out
  0 1 2 3 4 5 6 7 8 9
```

## `ran`: Generate a Random Number between 0 and 1

Repeated calls to `ran` generate a sequence of random numbers with a uniform distribution.

| *r* = ran( *i* ) | | | |
|---|---|---|---|
| *i* | `INTEGER*4` | Input | Variable or array element |
| *r* | `REAL` | Output | Variable or array element |

See *lcrans*(3m).

Example: `ran`:

```
demo% cat ran1.f
* ran1.f -- Generate random numbers.
    INTEGER*4 i, n
    real r(10)
    i = 760013
    do n = 1, 10
        r(n) = ran ( i )
    end do
    write ( *, "( 5 f11.6 )" ) r
    end
demo% f77 -silent ran1.f
demo% a.out
   0.222058 0.299851 0.390777 0.607055 0.653188
   0.060174 0.149466 0.444353 0.002982 0.976519
demo%
```

Note the following:

- The range includes 0.0 and excludes 1.0.
- The algorithm is a multiplicative, congruential type, general random number generator.
- In general, the value of `i` is set *once* during execution of the calling program.
- The initial value of `i` should be a large odd integer.
- Each call to `RAN` gets the next random number in the sequence.

- To get a different sequence of random numbers each time you run the program, you must set the argument to a different initial value for each run.
- The argument is used by `RAN` to store a value for the calculation of the next random number according to the following algorithm:

```
SEED = 6909 * SEED + 1 (MOD 2**32)
```

- `SEED` contains a 32-bit number, and the high-order 24 bits are converted to floating point, and that value is returned.

## `rand`, `drand`, `irand`: Return Random Values

`rand` returns real values in the range 0.0 through 1.0.

`drand` returns double precision values in the range 0.0 through 1.0.

`irand` returns positive integers in the range 0 through 2147483647.

These functions use *random*(3) to generate sequences of random numbers. The three functions share the same 256 byte state array. The only advantage of these functions is that they are widely available on UNIX systems. For better random number generators, compare `lcrans`, `addrans`, and `shufrans`. See also *random*(3), and the *Numerical Computation Guide*

| ***i*** = `irand(` ***k*** `)`<br>***r*** = `rand(` ***k*** `)`<br>***d*** = `drand(` ***k*** `)` | | | |
|---|---|---|---|
| *k* | `INTEGER*4` | Input | *k*=0: Get next random number in the sequence<br>*k*=1: Restart sequence, return first number<br>*k*>0: Use as a seed for new sequence, return first<br>number |
| `rand` | `REAL*4` | Output | |

| | | | |
|---|---|---|---|
| ***i*** = `irand(` ***k*** `)`<br><br>***r*** = `rand(` ***k*** `)`<br><br>***d*** = `drand(` ***k*** `)` | | | |
| `drand` | `REAL*8` | Output | |
| `irand` | `INTEGER*4` | Output | |

Example: `irand()`:

```
    integer*4 v(5), iflag/0/
    do i = 1, 5
        v(i) = irand( iflag )
    end do
    write(*,*) v
    end
demo% f77 -silent trand.f
demo% a.out
   2078917053 143302914 1027100827 1953210302 755253631
demo%
```

## `rename`: Rename a File

The function is called by:

| | | | |
|---|---|---|---|
| `INTEGER*4 rename`<br><br>***status* = rename( *from, to* )** | | | |
| *from* | `character*`*n* | Input | Path name of an existing file |
| *to* | `character*`*n* | Input | New path name for the file |
| Return value | `INTEGER*4` | Output | *status*=0: OK<br><br>*status*>0: System error code |

If the file specified by *to* exists, then both *from* and *to* must be the same type of file, and must reside on the same file system. If *to* exists, it is removed first.

Example: `rename()`—Rename file `trename.old` to `trename.new`

```
demo% cat trename.f
    INTEGER*4 rename, status
    character*18 from/"trename.old"/, to/"trename.new"/
    status = rename( from, to )
    if ( status .ne. 0 ) stop "rename: error"
    end
demo% f77 -silent trename.f
demo% ls trename*
trename.f trename.old
demo% a.out
demo% ls trename*
trename.f trename.new
demo%
```

See also *rename*(2) and *perror*(3F).

Note: the path names cannot be longer than `MAXPATHLEN` as defined in `<sys/param.h>`.

## `secnds`: Get System Time in Seconds, Minus Argument

| *t* = secnds( *t0* ) | | | |
|---|---|---|---|
| *t0* | REAL | Input | Constant, variable, or array element |
| Return Value | REAL | Output | Number of seconds since midnight, minus *t0* |

Example: `secnds`:

```
demo% cat sec1.f
    real elapsed, t0, t1, x, y
    t0 = 0.0
    t1 = secnds( t0 )
    y = 0.1
    do i = 1, 1000
        x = asin( y )
    end do
    elapsed = secnds( t1 )
    write ( *, 1 ) elapsed
1    format ( " 1000 arcsines: ", f12.6, " sec" )
    end
```

```
demo% f77 -silent sec1.f
demo% a.out
 1000 arcsines: 6.699141 sec
demo%
```

Note that:

- The returned value from SECNDS is accurate to 0.01 second.
- The value is the system time, as the number of seconds from midnight, and it correctly spans midnight.
- Some precision may be lost for small time intervals near the end of the day.

## sh: Fast Execution of an sh Command

The function is called by:

| INTEGER*4 sh<br><br>***status*** = sh( ***string*** ) | | | |
|---|---|---|---|
| *string* | character**n* | Input | String containing command to do |
| Return value | INTEGER*4 | Output | Exit status of the shell executed. See *wait*(2) for an explanation of this value. |

Example: sh():

```
    character*18 string / "ls > MyOwnFile.names" /
    INTEGER*4 status, sh
    status = sh( string )
    if ( status .ne. 0 ) stop "sh: error"
    ...
    end
```

The function sh passes *string* to the sh shell as input, as if the string had been typed as a command.

The current process waits until the command terminates.

The forked process flushes all open files:

- For output files, the buffer is flushed to the actual file.
- For input files, the position of the pointer is unpredictable.

The `sh()` function is not MT-safe. Do not call it from multithreaded or parallelized programs.

See also: *execve*(2), *wait*(2), and *system*(3).

Note: *string* cannot be longer than 1,024 characters.

## `signal`: Change the Action for a Signal

The function is called by:

| `INTEGER*4 signal` *or* `INTEGER*8 signal`<br><br>*n* `= signal(` *signum*`,` *proc*`,` *flag* `)` | | | |
|---|---|---|---|
| *signum* | `INTEGER*4` | Input | Signal number; see *signal*(3) |
| *proc* | Routine name | Input | Name of user signal handling routine; must be in an external statement |
| *flag* | `INTEGER*4` | Input | *flag*<0: Use *proc* as the signal handler<br><br>*flag*>=0: Ignore *proc*; pass *flag* as the action:<br><br>*flag*=`0`: Use the default action<br><br>*flag*=1: Ignore this signal |
| Return value | `INTEGER*4`<br><br><br><br><br><br><br><br>`INTEGER*8` | Output | *n*=-1: System error<br><br>*n*>0: Definition of previous action<br><br>*n*>1: *n*=Address of routine that would have been called<br><br>*n*<-1: If *signum* is a valid signal number, then: *n*=address of routine that would have been called. If *signum* is a *not* a valid signal number, then: *n* is an error number.<br><br>On 64-bit environments, `signal` and the variables receiving its output must be declared `INTEGER*8` |

If *proc* is called, it is passed the signal number as an integer argument.

If a process incurs a signal, the default action is usually to clean up and abort. A signal handling routine provides the capability of catching specific exceptions or interrupts for special processing.

The returned value can be used in subsequent calls to `signal` to restore a previous action definition.

You can get a negative return value even though there is no error. In fact, if you pass a *valid* signal number to `signal()`, and you get a return value less than -1, then it is OK.

`f77` arranges to trap certain signals when a process is started. The only way to restore the default `f77` action is to save the returned value from the first call to `signal`.

`f77_floatingpoint.h` defines *proc* values `SIGFPE_DEFAULT`, `SIGFPE_IGNORE`, and `SIGFPE_ABORT`. See "`f77_floatingpoint.h`: FORTRAN IEEE Definitions" on page 39

In 64-bit environments, `signal` must be declared `INTEGER*8`, along with the variables receiving its output, to avoid truncation of the address that may be returned.

See also *kill*(1), *signal*(3), and *kill*(3F), and *Numerical Computation Guide.*

## `sleep`: Suspend Execution for an Interval

The subroutine is called by:

| call sleep( ***itime*** ) | | | |
|---|---|---|---|
| *itime* | `INTEGER*4` | Input | Number of seconds to sleep |

The actual time can be up to 1 second less than *itime* due to granularity in system timekeeping.

Example: `sleep()`:

```
    INTEGER*4 time / 5 /
    write(*,*) "Start"
    call sleep( time )
    write(*,*) "End"
    end
```

See also *sleep*(3).

## `stat`, `lstat`, `fstat`: Get File Status

These functions return the following information:

- device,
- inode number,
- protection,
- number of hard links,
- user ID,
- group ID,
- device type,
- size,
- access time,
- modify time,
- status change time,
- optimal blocksize,
- blocks allocated

Both `stat` and `lstat` query by file name. `fstat` queries by logical unit.

### `stat`: Get Status for File, by File Name

The function is called by:

| INTEGER*4 stat<br><br>*ierr* = stat ( *name*, *statb* ) | | | |
|---|---|---|---|
| *name* | character**n* | Input | Name of the file |
| *statb* | INTEGER*4 | Output | Status structure for the file,<br>13-element array |
| Return value | INTEGER*4 | Output | *ierr*=0: OK<br><br>*ierr*>0: Error code |

Example 1: `stat()`:

```
character name*18 /"MyFile"/
INTEGER*4 ierr, stat, lunit/1/, statb(13)
open( unit=lunit, file=name )
ierr = stat ( name, statb )
if ( ierr .ne. 0 ) stop "stat: error"
write(*,*)"UID of owner = ",statb(5),", blocks = ",statb(13)
end
```

## `fstat`: Get Status for File, by Logical Unit

The function

| INTEGER*4 fstat<br><br>*ierr* = fstat ( *lunit*, *statb* ) | | | |
|---|---|---|---|
| *lunit* | INTEGER*4 | Input | Logical unit number |
| *statb* | INTEGER*4 | Output | Status for the file: 13-element array |
| Return value | INTEGER*4 | Output | *ierr*=0: OK<br><br>*ierr*>0: Error code |

is called by:

Example 2: `fstat()`:

```
character name*18 /"MyFile"/
INTEGER*4 fstat, lunit/1/, statb(13)
open( unit=lunit, file=name )
ierr = fstat ( lunit, statb )
if ( ierr .ne. 0 ) stop "fstat: error"
write(*,*)"UID of owner = ",statb(5),", blocks = ",statb(13)
end
```

## `lstat`: Get Status for File, by File Name

The function is called by:

| *ierr* = `lstat` ( *name*, *statb* ) | | | |
|---|---|---|---|
| *name* | `character*`*n* | Input | File name |
| *statb* | `INTEGER*4` | Output | Status array of file, 13 elements |
| Return value | `INTEGER*4` | Output | *ierr*=0: OK<br>*ierr*>0: Error code |

Example 3: `lstat()`:

```
character name*18 /"MyFile"/
INTEGER*4 lstat, lunit/1/, statb(13)
open( unit=lunit, file=name )
ierr = lstat ( name, statb )
if ( ierr .ne. 0 ) stop "lstat: error"
write(*,*)"UID of owner = ",statb(5),", blocks = ",statb(13)
end
```

## Detail of Status Array for Files

The meaning of the information returned in the INTEGER*4 array *statb* is as described for the structure *stat* under *stat*(2).

Spare values are not included. The order is shown in the following table:

| | |
|---|---|
| `statb`(1) | Device inode resides on |
| `statb`(2) | This inode's number |
| `statb`(3) | Protection |
| `statb`(4) | Number of hard links to the file |
| `statb`(5) | User ID of owner |
| `statb`(6) | Group ID of owner |
| `statb`(7) | Device type, for inode that is device |
| `statb`(8) | Total size of file |
| `statb`(9) | File last access time |
| `statb`(10) | File last modify time |
| `statb`(11) | File last status change time |
| `statb`(12) | Optimal blocksize for file system I/O ops |
| `statb`(13) | Actual number of blocks allocated |

See also *stat*(2), *access*(3F), *perror*(3F), and *time*(3F).

Note: the path names can be no longer than `MAXPATHLEN` as defined in `<sys/param.h>`.

## `stat64`, `lstat64`, `fstat64`: Get File Status

64-bit "long file" (Solaris 2.6 and Solaris 7) versions of stat, lstat, fstat. These routines are identical to the non-64-bit routines, except that the 13-element array *statb* must be declared `INTEGER*8`.

## `system`: Execute a System Command

The function is called by:

| INTEGER*4 system<br><br>***status*** = system( ***string*** ) | | | |
|---|---|---|---|
| *string* | character**n* | Input | String containing command to do |
| Return value | INTEGER*4 | Output | Exit status of the shell executed. See *wait*(2) for an explanation of this value. |

Example: `system()`:

```
    character*8 string / "ls s*" /
    INTEGER*4 status, system
    status = system( string )
    if ( status .ne. 0 ) stop "system: error"
    end
```

The function `system` passes *string* to your shell as input, as if the string had been typed as a command. Note: *string* cannot be longer than 1024 characters.

If `system` can find the environment variable `SHELL`, then `system` uses the value of `SHELL` as the command interpreter (shell); otherwise, it uses *sh*(1).

The current process waits until the command terminates.

Historically, `cc` and `f77` developed with different assumptions:

- If `cc` calls `system`, the shell is always the Bourne shell.
- If `f77` calls `system`, then which shell is called depends on the environment variable `SHELL`.

The `system` function flushes all open files:

- For output files, the buffer is flushed to the actual file.
- For input files, the position of the pointer is unpredictable.

See also: *execve*(2), *wait*(2), and *system*(3).

The `system()` function is not MT-safe. Do not call it from multithreaded or parallelized programs.

## `time`, `ctime`, `ltime`, `gmtime`: Get System Time

These routines have the following functions:

| | |
|---|---|
| `time` | Standard version: Get system time as integer (seconds since 0 GMT 1/1/70)VMS Version: Get the system time as character (hh:mm:ss) |
| `ctime` | Convert a system time to an ASCII string. |
| `ltime` | Dissect a system time into month, day, and so forth, local time. |
| `gmtime` | Dissect a system time into month, day, and so forth, GMT. |

## `time`: Get System Time

For `time()`, there are two versions, a standard version and a VMS version. If you use the `f77` command-line option `-lV77`, then you get the VMS version for `time()` and for `idate()`; otherwise, you get the standard versions.

The standard function is called by:

| `INTEGER*4 time` ***or*** `INTEGER*8`<br>***n*** = `time()` ***Standard Version*** | | | |
|---|---|---|---|
| Return value | `INTEGER*4` | Output | Time, in seconds, since 0:0:0, GMT, 1/1/70 |
| | `INTEGER*8` | Output | In 64-bit environments, `time` returns an INTEGER*8 value |

The function `time`() returns an integer with the time since 00:00:00 GMT, January 1, 1970, measured in seconds. This is the value of the operating system clock.

Example: `time()`, version standard with the operating system:

```
INTEGER*4  n, time
n = time()
write(*,*) "Seconds since 0 1/1/70 GMT = ", n
```

```
    end
demo% f77 -silent ttime.f
demo% a.out
 Seconds since 0 1/1/70 GMT =   913240205
demo%
```

The VMS version of `time` is a subroutine that gets the current system time as a character string.

The VMS subroutine is called by:

| call time( *t* ) ***VMS Version*** | | | |
|---|---|---|---|
| *t* | `character*8` | Output | Time, in the form *hh*:*mm*:*ss hh*, *mm*, and *ss* are each two digits: *hh* is the hour; *mm* is the minute; *ss* is the second |

Example: `time(`*t*`)`, VMS version, `ctime`—convert the system time to ASCII:

```
    character  t*8
    call time( t )
    write(*, "(" The current time is ", A8 )")  t
    end
demo% f77 -silent ttimeV.f -lV77
demo% a.out
 The current time is 08:14:13
demo%
```

## `ctime`: Convert System Time to Character

The function `ctime` converts a system time, *stime*, and returns it as a 24-character ASCII string.

The function is called by:

| CHARACTER ctime*24<br><br>***string*** = ctime( ***stime*** ) | | | |
|---|---|---|---|
| *stime* | INTEGER*4 | Input | System time from time() (standard version) |
| Return value | character*24 | Output | System time as character string. Declare ctime and *string* as character*24. |

The format of the ctime returned value is shown in the following example. It is described in the man page *ctime*(3C).

Example: ctime():

```
    character*24 ctime, string
    INTEGER*4  n, time
    n = time()
    string = ctime( n )
    write(*,*) "ctime: ", string
    end
demo% f77 -silent tctime.f
demo% a.out
 ctime: Wed Dec  9 13:50:05 1998
demo%
```

## ltime: Split System Time to Month, Day,… (Local)

This routine dissects a system time into month, day, and so forth, for the local time zone.

The subroutine is called by:

| call ltime( ***stime***, ***tarray*** ) | | | |
|---|---|---|---|
| *stime* | INTEGER*4 | Input | System time from time() (standard version) |
| *tarray* | INTEGER*4(9) | Output | System time, local, as day, month, year, … |

For the meaning of the elements in tarray, see the next section.

Example: `ltime()`:

```
    integer*4  stime, tarray(9), time
    stime = time()
    call ltime( stime, tarray )
    write(*,*) "ltime: ", tarray
    end
demo% f77 -silent tltime.f
demo% a.out
 ltime: 25 49 10 12 7 91 1 223 1
demo%
```

## `gmtime`: Split System Time to Month, Day, … (GMT)

This routine dissects a system time into month, day, and so on, for GMT.

The subroutine is:

| call gmtime( ***stime***, ***tarray*** ) | | | |
|---|---|---|---|
| *stime* | `INTEGER*4` | Input | System time from `time()` (standard version) |
| *tarray* | `INTEGER*4(9)` | Output | System time, GMT, as day, month, year, … |

Example: `gmtime`:

```
    integer*4  stime, tarray(9), time
    stime = time()
    call gmtime( stime, tarray )
    write(*,*) "gmtime: ", tarray
    end
demo% f77 -silent tgmtime.f
demo% a.out
 gmtime:   12  44  19  18  5  94  6  168  0
demo%
```

Here are the `tarray()` values for `ltime` and `gmtime`: index, units, and range:

| | |
|---|---|
| 1 Seconds (0 - 61) | 6 Year - 1900 |
| 2 Minutes (0 - 59) | 7 Day of week (Sunday = 0) |
| 3 Hours (0 - 23) | 8 Day of year (0 - 365) |
| 4 Day of month (1 - 31) | 9 Daylight Saving Time, 1 if DST in effect |
| 5 Months since January (0 - 11) | |

These values are defined by the C library routine *ctime*(3C), which explains why the system may return a count of seconds *greater than* 59. See also: *idate*(3F), and *fdate*(3F).

### `ctime64`, `gmtime64`, `ltime64`: System Time Routines for 64-bit Environments

These are versions of the corresponding routines `ctime`, `gmtime`, and `ltime`, to provide portability on 64-bit environments. They are identical to these routines except that the input variable *stime* must be INTEGER*8.

When used in a 32-bit environment with an `INTEGER*8` *stime*, if the value of *stime* is beyond the INTEGER*4 range `ctime64` returns all asterisks, while `gmtime` and `ltime` fill the tarray array with -1.

## `topen`, `tclose`, `tread`,..., `tstate`: Tape I/O

*(FORTRAN 77 Only)* These routines provide an alternative way to manipulate magnetic tape:

| | |
|---|---|
| `topen` | Associate a device name with a tape logical unit. |
| `tclose` | Write EOF, close tape device channel, and remove association with *tlu*. |
| `tread` | Read next physical record from tape into buffer. |
| `twrite` | Write the next physical record from buffer to tape. |
| `trewin` | Rewind the tape to the beginning of the first data file. |

| | |
|---|---|
| `tskipf` | Skip forward over files and/or records, and reset EOF status. |
| `tstate` | Determine the logical state of the tape I/O channel. |

On any one unit, do not mix these functions with standard FORTRAN I/O.

You must first use `topen`() to open a tape logical unit, *tlu*, for the specified device. Then you do all other operations on the specified *tlu*. *tlu* has no relationship at all to any normal FORTRAN logical unit.

Before you use one of these functions, its name must be in an `INTEGER*4` type statement.

## `topen`: Associate a Device with a Tape Logical Unit

The function is called by:

| `INTEGER*4 topen`<br>***n*** `= topen(` ***tlu***`,` ***devnam***`,` ***islabeled*** `)` | | | |
|---|---|---|---|
| *tlu* | `INTEGER*4` | Input | Tape logical unit, in the range 0 to 7. |
| devnam | CHARACTER | Input | Device name; for example: `'/dev/rst0'` |
| *islabeled* | `LOGICAL` | Input | True=the tape is labeled<br>A label is the first file on the tape. |
| Return value | `INTEGER*4` | Output | *n*=0: OK<br>*n*<0: Error |

This function does *not* move the tape. See *perror*(3F) for details.

Example: `topen()`—open a 1/4-inch tape file:

```
CHARACTER devnam*9 / "/dev/rst0" /
INTEGER*4 n / 0 /, tlu / 1 /, topen
LOGICAL islabeled / .false. /
```

```
    n = topen( tlu, devnam, islabeled )
    IF ( n .LT. 0 ) STOP "topen: cannot open"
    WRITE(*,"("topen ok:", 2I3, 1X, A10)") n, tlu,  devnam
    END
```

The output is:

```
topen ok: 0 1 /dev/rst0
```

## `tclose`: Write EOF, Close Tape Channel, Disconnect *tlu*

The function is called by:

| INTEGER*4 tclose<br><br>***n*** = tclose ( ***tlu*** ) | | | |
|---|---|---|---|
| *tlu* | INTEGER*4 | Input | Tape logical unit, in range 0 to 7 |
| *n* | INTEGER*4 | Return value | *n*=0: OK<br>*n*<0: Error |

**Caution -** `tclose()` places an EOF marker immediately after the current location of the unit pointer, and then closes the unit. So if you `trewin()` a unit before you `tclose()` it, its contents are discarded.

Example: `tclose()`—close an opened 1/4-inch tape file:

```
    CHARACTER devnam*9 / "/dev/rst0" /
    INTEGER*4 n / 0 /, tlu / 1 /, tclose, topen
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    n = tclose( tlu )
    IF ( n .LT. 0 ) STOP "tclose: cannot close"
    WRITE(*, "("tclose ok:", 2I3, 1X, A10)")  n, tlu,  devnam
    END
```

The output is:

```
tclose ok: 0 1  /dev/rst0
```

## `twrite`: Write Next Physical Record to Tape

The function is called by:

| INTEGER*4 twrite<br>n = twrite( ***tlu***, ***buffer*** ) | | | |
|---|---|---|---|
| *tlu* | `INTEGER*4` | Input | Tape logical unit, in range 0 to 7 |
| *buffer* | `character` | Input | Must be sized at a multiple of 512 |
| *n* | `INTEGER*4` | Return value | *n*>0: OK, and *n* = the number of bytes written<br>*n*=0: End of Tape<br>*n*<0: Error |

The physical record length is the size of `buffer`.

Example: `twrite()`—write a 2-record file:

```
    CHARACTER devnam*9 / "/dev/rst0" /, rec1*512 / "abcd" /,
&            rec2*512 / "wxyz" /
    INTEGER*4 n / 0 /, tlu / 1 /, tclose, topen, twrite
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    IF ( n .LT. 0 ) STOP "topen: cannot open"
    n = twrite( tlu, rec1 )
    IF ( n .LT. 0 ) STOP "twrite: cannot write 1"
    n = twrite( tlu, rec2 )
    IF ( n .LT. 0 ) STOP "twrite: cannot write 2"
    WRITE(*, "("twrite ok:", 2I4, 1X, A10)")  n, tlu, devnam
    END
```

The output is:

```
twrite ok: 512 1 /dev/rst0
```

## `tread`: Read Next Physical Record from Tape

The function is called by:

| `INTEGER*4 tread`<br>***n*** = `tread(` ***tlu***, ***buffer*** `)` | | | |
|---|---|---|---|
| *tlu* | `INTEGER*4` | Input | Tape logical unit, in range 0 to 7. |
| *buffer* | `character` | Input | Must be sized at a multiple of 512, and must be large enough to hold the largest physical record to be read. |
| *n* | `INTEGER*4` | Return value | *n*>0: OK, and *n* is the number of bytes read.<br>*n*<0: Error<br>*n*=0: EOF |

If the tape is at EOF or EOT, then `tread` does a return; it does not read tapes.

Example: `tread()`—read the first record of the file written above:

```
    CHARACTER devnam*9 / "/dev/rst0" /, onerec*512 / " " /
    INTEGER*4 n / 0 /, tlu / 1 /, topen, tread
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    IF ( n .LT. 0 ) STOP "topen: cannot open"
    n = tread( tlu, onerec )
    IF ( n .LT. 0 ) STOP "tread: cannot read"
    WRITE(*,"("tread ok:", 2I4, 1X, A10)")  n, tlu,  devnam
    WRITE(*,"( A4)")  onerec
    END
```

The output is:

```
tread ok: 512 1 /dev/rst0
abcd
```

## `trewin`: Rewind Tape to Beginning of First Data File

The function is called by:

| | | |
|---|---|---|
| `INTEGER*4 trewin`<br>***n*** = `trewin (` ***tlu*** `)` | | |
| *tlu* `INTEGER*4` | Input | Tape logical unit, in range 0 to 7 |
| *n* `INTEGER*4` | Return value | *n*=0: OK<br>*n*<0: Error |

If the tape is labeled, then the label is skipped over after rewinding.

Example 1: `trewin()`—typical fragment:

```
    CHARACTER devnam*9 / "/dev/rst0" /
    INTEGER*4 n /0/, tlu /1/, tclose, topen, tread, trewin
    …
    n = trewin( tlu )
    IF ( n .LT. 0 ) STOP "trewin: cannot rewind"
    WRITE(*, "("trewin ok:", 2I4, 1X, A10)") n, tlu, devnam
    …
    END
```

Example 2: `trewin()`—in a two-record file, try to read three records, rewind, read one record:

```
    CHARACTER devnam*9 / "/dev/rst0" /, onerec*512 / " " /
    INTEGER*4 n / 0 /, r, tlu / 1 /, topen, tread, trewin
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    IF ( n .LT. 0 ) STOP "topen: cannot open"
    DO r = 1, 3
       n = tread( tlu, onerec )
       WRITE(*,"(1X, I2, 1X, A4)")  r, onerec
    END DO
    n = trewin( tlu )
    IF ( n .LT. 0 ) STOP "trewin: cannot rewind"
    WRITE(*, "("trewin ok:" 2I4, 1X, A10)")  n, tlu, devnam
    n = tread( tlu, onerec )
    IF ( n .LT. 0 ) STOP "tread: cannot read after rewind"
    WRITE(*,"(A4)")  onerec
    END
```

The output is:

```
1 abcd
2 wxyz
3 wxyz
trewin ok: 0 1 /dev/rst0
abcd
```

## `tskipf`: Skip Files and Records; Reset EoF Status

The function is called by:

| | | |
|---|---|---|
| `INTEGER*4 tskipf`<br><br>***n*** = `tskipf(` ***tlu***, ***nf***, ***nr*** `)` | | |
| *tlu* `INTEGER*4` | Input | Tape logical unit, in range 0 to 7 |
| *nf* `INTEGER*4` | Input | Number of end-of-file marks to skip over first |
| *nr* `INTEGER*4` | Input | Number of physical records to skip over after skipping files |
| *n* `INTEGER*4` | Return value | *n*=0: OK<br><br>*n*<0: Error |

This function does *not* skip backward.

First, the function skips forward over *nf* end-of-file marks. Then, it skips forward over *nr* physical records. If the current file is at EOF, this counts as one file to skip. This function also resets the EOF status.

Example: `tskipf()`—typical fragment: skip four files and then skip one record:

```
INTEGER*4 nfiles / 4 /, nrecords / 1 /, tskipf, tlu / 1 /
…
n = tskipf( tlu, nfiles, nrecords )
IF ( n .LT. 0 ) STOP "tskipf: cannot skip"
…
```

Compare with `tstate`() .

## `tstate`: Get Logical State of Tape I/O Channel

The function is called by:

| `INTEGER*4 tstate`<br>***n*** = `tstate(` ***tlu***, ***fileno***, ***recno***, ***errf***, ***eoff***, ***eotf***, ***tcsr*** `)` | | | |
|---|---|---|---|
| *tlu* | `INTEGER*4` | Input | Tape logical unit, in range 0 to 7 |
| *fileno* | `INTEGER*4` | Output | Current file number |
| *recno* | `INTEGER*4` | Output | Current record number |
| *errf* | `LOGICAL` | Output | True=an error occurred |
| *eoff* | `LOGICAL` | Output | True=the current file is at EOF |
| *eotf* | `LOGICAL` | Output | True=tape has reached logical end-of-tape |
| *tcsr* | `INTEGER*4` | Output | True=hardware errors on the device. It contains the tape drive control status register. If the error is software, then *tcsr* is returned as zero. The values returned in this status register vary grossly with the brand and size of tape drive. |

For details, see *st*(4s).

While *eoff* is true, you cannot read from that *tlu*. You can set this EOF status flag to false by using `tskipf()` to skip one file and zero records:

```
    n = tskipf( tlu, 1, 0).
```

Then you can read any valid record that follows.

End-of-tape (EOT) is indicated by an empty file, often referred to as a double `EOF` mark. You cannot read past EOT, but you can write past it.

Example: Write three files of two records each:

```
    CHARACTER devnam*10 / "/dev/nrst0" /,
&                 f0rec1*512 / "eins" /, f0rec2*512 / "zwei" /,
&                 f1rec1*512 / "ichi" /, f1rec2*512 / "ni__" /,
&                 f2rec1*512 / "un__" /, f2rec2*512 / "deux" /
    INTEGER*4 n / 0 /, tlu / 1 /, tclose, topen, trewin, twrite
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    n = trewin( tlu )
    n = twrite( tlu, f0rec1 )
    n = twrite( tlu, f0rec2 )
    n = tclose( tlu )
    n = topen( tlu, devnam, islabeled )
    n = twrite( tlu, f1rec1 )
    n = twrite( tlu, f1rec2 )
    n = tclose( tlu )
    n = topen( tlu, devnam, islabeled )
    n = twrite( tlu, f2rec1 )
    n = twrite( tlu, f2rec2 )
    n = tclose( tlu )
    END
```

The next example uses `tstate()` to trap EOF and get at all files.

Example: Use `tstate()` in a loop that reads all records of the 3 files written in the previous example:

```
    CHARACTER devnam*10 / "/dev/nrst0" /, onerec*512 / " " /
    INTEGER*4 f, n / 0 /, tlu / 1 /, tcsr, topen, tread,
&        trewin, tskipf, tstate
    LOGICAL errf, eoff, eotf, islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
    WRITE(*,1)  "open:", fn, rn, errf, eoff, eotf, tcsr
1    FORMAT(1X, A10, 2I2, 1X, 1L, 1X, 1L,1X, 1L, 1X, I2 )
2    FORMAT(1X, A10,1X,A4,1X,2I2,1X,1L,1X,1L,1X,1L,1X,I2)
    n = trewin( tlu )
    n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
    WRITE(*,1)  "rewind:", fn, rn, errf, eoff, eotf, tcsr
    DO f = 1, 3
       eoff = .false.
       DO WHILE ( .NOT. eoff )
          n = tread( tlu, onerec )
          n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
          IF (.NOT. eoff) WRITE(*,2) "read:", onerec,
&            fn, rn, errf, eoff, eotf, tcsr
       END DO
       n = tskipf( tlu, 1, 0 )
       n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
       WRITE(*,1)  "tskip: ", fn, rn, errf, eoff, eotf, tcsr
    END DO
    END
```

The output is:

```
open: 0 0 F F F 0
rewind: 0 0 F F F 0
read: eins 0 1 F F F 0
read: zwei 0 2 F F F 0
tskip: 1 0 F F F 0
read: ichi 1 1 F F F 0
read: ni__ 1 2 F F F 0
tskip: 2 0 F F F 0
read: un__ 2 1 F F F 0
read: deux 2 2 F F F 0
tskip: 3 0 F F F 0
```

A summary of EOF and EOT follows:

- If you are at either EOF or EOT, then:
  - Any `tread()` just returns; it does not read the tape.
  - A successful `tskipf(tlu,1,0)` resets the EOF status to false, and returns; it does not advance the tape pointer.
- A successful `twrite()` resets the EOF and EOT status flags to false.
- A successful `tclose()` resets all those flags to false.
- `tclose()` truncates; it places an EOF marker immediately after the current location of the unit pointer, and then closes the unit. So, if you use `trewin()` to rewind a unit before you use `tclose()` to close it, its contents are discarded. This behavior of `tclose()` is inherited from the Berkeley code.

See also: *ioctl*(2), *mtio*(4s), *perror*(3F), *read*(2), *st*(4s), and *write*(2).

## `ttynam`, `isatty`: Get Name of a Terminal Port

`ttynam` and `isatty` handle terminal port names.

### `ttynam`: Get Name of a Terminal Port

The function `ttynam` returns a blank padded path name of the terminal device associated with logical unit *lunit*.

The function is called by:

| CHARACTER ttynam*24<br><br>***name*** = ttynam( ***lunit*** ) | | | |
|---|---|---|---|
| *lunit* | INTEGER*4 | Input | Logical unit |
| Return value | character**n* | Output | If nonblank returned: *name*=path name of device on *lunit*. Size *n* must be large enough for the longest path name.<br><br>If empty string (all blanks) returned: *lunit* is not associated with a terminal device in the directory, /dev |

## isatty: Is this Unit a Terminal?

The function

| ***terminal*** = isatty( ***lunit*** ) | | | |
|---|---|---|---|
| *lunit* | INTEGER*4 | Input | Logical unit |
| Return value | LOGICAL*4 | Output | *terminal*=true: It is a terminal device<br><br>*terminal*=false: It is *not* a terminal device |

is called by:

Example: Determine if *lunit* is a tty:

```
character*12 name, ttynam
INTEGER*4 lunit /5/
logical*4 isatty, terminal
terminal = isatty( lunit )
name = ttynam( lunit )
write(*,*) "terminal = ", terminal, ", name = "", name, """
end
```

The output is:

```
terminal = T, name = "/dev/ttyp1  "
```

## unlink: Remove a File

The function is called by:

| INTEGER*4 unlink<br><br>***n*** = unlink ( ***patnam*** ) | | | |
|---|---|---|---|
| *patnam* | character**n* | Input | File name |
| Return value | INTEGER*4 | Output | *n*=0: OK<br>*n*>0: Error |

The function unlink removes the file specified by path name *patnam*. If this is the last link to the file, the contents of the file are lost.

Example: unlink()—Remove the tunlink.data file:

```
    call unlink( "tunlink.data" )
    end
demo% f77 -silent tunlink.f
demo% ls tunl*
tunlink.f tunlink.data
demo% a.out
demo% ls tunl*
tunlink.f
demo%
```

See also: *unlink*(2), *link*(3F), and *perror*(3F). Note: the path names cannot be longer than MAXPATHLEN as defined in <sys/param.h>.

## wait: Wait for a Process to Terminate

The function is:

| INTEGER*4 wait<br><br>n = wait( ***status*** ) | | | |
|---|---|---|---|
| *status* | INTEGER*4 | Output | Termination status of the child process |
| Return value | INTEGER*4 | Output | *n*>0: Process ID of the child process<br><br>*n*<0: *n*=System error code; see *wait*(2). |

`wait` suspends the caller until a signal is received, or one of its child processes terminates. If any child has terminated since the last `wait`, return is immediate. If there are no children, return is immediate with an error code.

Example: Code fragment using `wait()`:

```
INTEGER*4 n, status, wait
…
n = wait( status )
if ( n .lt. 0 ) stop 'wait: error'
…
end
```

See also: *wait*(2), *signal*(3F), *kill*(3F), and *perror*(3F).

# Index