Fortran Library Reference

Sun™ Studio 10

Sun Microsystems, Inc.
www.sun.com

Part No. 819-0490-10
January 2005, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Contents

Tables

Before You Begin

The *Fortran Library Reference* describes the intrinsic functions and routines in the Sun™ Studio Fortran libraries. This reference manual is intended for programmers with a working knowledge of the Fortran language and the Solaris™ operating environment.

This guide is intended for scientists, engineers, and programmers who have a working knowledge of the Fortran language and wish to learn how to use the Sun Fortran compilers effectively. Familiarity with the Solaris operating environment or UNIX® in general is also assumed.

Discussion of Fortran programming issues on Solaris operating environments, including input/output, application development, library creation and use, program analysis, porting, optimization, and parallelization can be found in the companion *Fortran Programming Guide*.

Typographic Conventions

TABLE P-1 Typeface Conventions

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

- The symbol Δ stands for a blank space where a blank is significant:

```
ΔΔ36.001
```

- The FORTRAN 77 standard used an older convention, spelling the name "FORTRAN" capitalized. The current convention is to use lower case: "Fortran 95"

- References to online man pages appear with the topic name and section number. For example, a reference to the library routine `GETENV` will appear as `getenv`(3F), implying that the man command to access this man page would be:
  `man -s 3F getenv`

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*`n`*`]`	`-O4, -O`
{ }	Braces contain a set of choices for a required option.	`d{y|n}`	`-dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`-Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*`dir`*`[:`*`dir`*`]`	`-R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`-xinline=`*`f1`*`[,…`*`fn`*`]`	`-xinline=alpha,dos`

Shell Prompts

Shell	Prompt
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell, Korn shell, and GNU Bourne-Again shell	`$`
Superuser for Bourne shell, Korn shell, and GNU Bourne-Again shell	`#`

Supported Platforms

This Sun Studio release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris Operating System you are running are available in the hardware compatibility lists at `http://www.sun.com/bigadmin/hcl`. These documents cite any implementation differences between the platform types.

In this document, the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Software and Man Pages

The Sun Studio software and its man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the software, you must have your `PATH` environment variable set correctly (see "Accessing the Compilers and Tools" on page xiv). To access the man pages, you must have the your `MANPATH` environment variable set correctly (see "Accessing the Man Pages" on page xv.).

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), `ksh`(1), and `bash`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` variable and `MANPATH` variables to access this release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Sun Studio software is installed in the `/opt` directory. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Accessing the Compilers and Tools

Use the steps below to determine whether you need to change your `PATH` variable to access the compilers and tools.

▼ To Determine Whether You Need to Set Your PATH Environment Variable

1. **Display the current value of the PATH variable by typing the following at a command prompt.**

```
% echo $PATH
```

2. **Review the output to find a string of paths that contain /opt/SUNWspro/bin/.**

   If you find the path, your PATH variable is already set to access the compilers and tools. If you do not find the path, set your PATH environment variable by following the instructions in the next procedure.

▼ To Set Your PATH Environment Variable to Enable Access to the Compilers and Tools

1. **If you are using the C shell, edit your home .cshrc file. If you are using the Bourne shell or Korn shell, edit your home .profile file.**
2. **Add the following to your PATH environment variable. If you have Forte Developer software, Sun ONE Studio software or another release of Sun Studio software installed, add the following path before the paths to those installations.**

   /opt/SUNWspro/bin

Accessing the Man Pages

Use the following steps to determine whether you need to change your MANPATH variable to access the man pages.

▼ To Determine Whether You Need to Set Your MANPATH Environment Variable

1. **Request the dbx man page by typing the following at a command prompt.**

```
% man dbx
```

2. **Review the output, if any.**

   If the dbx(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next procedure for setting your MANPATH environment variable.

▼ To Set Your MANPATH Environment Variable to Enable Access to the Man Pages

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**
2. **Add the following to your `MANPATH` environment variable.**

   ```
   /opt/SUNWspro/man
   ```

Accessing the Integrated Development Environment

The Sun Studio integrated development environment (IDE) provides modules for creating, editing, building, debugging, and analyzing the performance of a C, C++, or Fortran application.

The command to start the IDE is `sunstudio`. For details on this command, see the `sunstudio(1)` man page.

The correct operation of the IDE depends on the IDE being able to find the core platform. The `sunstudio` command looks for the core platform in two locations:

- The command looks first in the default installation directory, `/opt/netbeans/3.5V`.
- If the command does not find the core platform in the default directory, it assumes that the directory that contains the IDE and the directory that contains the core platform are both installed in or mounted to the same location. For example, if the path to the directory that contains the IDE is `/foo/SUNWspro`, the command looks for the core platform in `/foo/netbeans/3.5V`.

If the core platform is not installed or mounted to either of the locations where the `sunstudio` command looks for it, then each user on a client system must set the environment variable `SPRO_NETBEANS_HOME` to the location where the core platform is installed or mounted (*/installation_directory*`/netbeans/3.5V`).

Each user of the IDE also must add */installation_directory*`/SUNWspro/bin` to their `$PATH` in front of the path to any other release of Forte Developer software, Sun ONE Studio software, or Sun Studio software.

The path */installation_directory*`/netbeans/3.5V/bin` should not be added to the user's `$PATH`.

Accessing Compilers and Tools Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html`.

  If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- Most manuals are available from the `docs.sun.com`sm web site. The following titles are available through your installed software only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*
- The release notes are available from the `docs.sun.com` web site.
- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialogs, in the IDE.

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index that is installed with the software on your local system or network.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed software on Solaris platforms through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Online help	HTML available through the Help menu in the IDE
Release notes	HTML at `http://docs.sun.com`

Related Compilers and Tools Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html` and `http://docs.sun.com`. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
Fortran Programming Guide	Describes how to write effective Fortran programs on Solaris environments; input/output, libraries, performance, debugging, and parallelization.

Document Title	Description
Fortran User's Guide	Describes the compile-time environment and command-line options for the `f95` compiler.
OpenMP API User's Guide	Summary of the OpenMP multiprocessing API, with specifics about the implementation.
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating system.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Resources for Developers

Visit `http://developers.sun.com/prodtech/cc` to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips

- Documentation of the software, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

You can find additional resources for developers at http://developers.sun.com.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

http://www.sun.com/service/contacting

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Send your comments about this document at:

http://www.sun.com/hwdocs/feedback

Please include the part number (819-0490-10) of your document in your comment.

CHAPTER 1

Fortran Library Routines

This chapter describes the Fortran library routines.

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`.

This chapter does not describe the standard Fortran 95 intrinsic routines. See the relevant Fortran 95 standards documents for information on intrinsics.

See also the *Numerical Computation Guide* for additional math routines that are callable from Fortran and C. These include the standard math library routines in `libm` and `libsunmath` (see `Intro`(3M)), optimized versions of these libraries, the SPARC vector math library, `libmvec`, and others.

See Chapter 3 for details on Fortran 77 and VMS intrinsic functions implemented by the `f95` compiler.

1.1 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 `-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 `-xtypemap` 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 `-xtypemap` and `-dbl` compiler options are used. Without explicit typing, these options could produce unexpected results. See the *Fortran User's Guide* and the `f95`(1) man page for details on these options.

The Fortran 95 compiler, `f95`, provides an include file, `system.inc`, that defines the interfaces for most non-intrinsic library routines. Include this file to insure that functions you call and their arguments are properly typed, especially when default data types are changed with `-xtypemap`.

```
include 'system.inc'
integer(4) mypid
mypid = getpid()
print *, mypid
```

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 `f95` compiler is described in the *Fortran User's Guide*, the *Fortran Programming Guide*, and the `f95`(1) man page.

1.2 64-Bit Environments

Compiling a program to run in a 64-bit operating environment (that is, compiling with `-xarch=v9`, `v9a`, or `v9b` and running the executable on a SPARC platform running the 64-bit enabled Solaris operating environment) changes the return values of certain functions. These are usually functions that interface standard system-level routines, such as `malloc`(3F) (see Section 1.4.35, "`malloc`, `malloc64`, `realloc`, `free`: Allocate/Reallocate/Deallocate Memory" on page 1-55), 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

Function	Description
`malloc64`	Allocate memory and return a pointer
`fseeko64`	Reposition a large file
`ftello64`	Determine position of a large file
`stat64, fstat64, lstat64`	Determine status of a file
`time64, ctime64, gmtime64, ltime64`	Dissect system time, convert to character
`qsort64`	Sort the elements of an array

1.3 Fortran Math Functions

The following functions and subroutines are part of the Fortran math libraries. They are available to all programs compiled with `f95`. These routines are non-intrinsics that take a specific data type as an argument and return the same. Non-intrinsics do have to be declared in the routine referencing them.

Many of these routines are "wrappers", Fortran interfaces to routines in the C language library, and as such are non-standard Fortran. They include IEEE recommended support functions, and specialized random number generators. See the *Numerical Computation Guide* and the man pages `libm_single`(3F), `libm_double`(3F), `libm_quadruple`(3F), for more information about these libraries.

1.3.1 Single-Precision Functions

These subprograms are single-precision math functions and subroutines.

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

These functions need not be explicitly typed with a REAL statement as long as default typing holds. (Names 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 1-2 Single-Precision Math Functions

Function Name	Return Type	Description
r_acos(x)	REAL	arc cosine
r_acosd(x)	REAL	--
r_acosh(x)	REAL	arc cosh
r_acosp(x)	REAL	--
r_acospi(x)	REAL	--
r_atan(x)	REAL	arc tangent
r_atand(x)	REAL	--
r_atanh(x)	REAL	arc tanh
r_atanp(x)	REAL	--
r_atanpi(x)	REAL	--
r_asin(x)	REAL	arc sine
r_asind(x)	REAL	--
r_asinh(x)	REAL	arc sinh
r_asinp(x)	REAL	--
r_asinpi(x)	REAL	--
r_atan2((y, x)	REAL	arc tangent
r_atan2d(y, x)	REAL	--
r_atan2pi(y, x)	REAL	--
r_cbrt(x)	REAL	cube root
r_ceil(x)	REAL	ceiling
r_copysign(x, y)	REAL	--
r_cos(x)	REAL	cosine
r_cosd(x)	REAL	--
r_cosh(x)	REAL	hyperb cos
r_cosp(x)	REAL	--
r_cospi(x)	REAL	--
r_erf(x)	REAL	err function
r_erfc(x)	REAL	--
r_expm1(x)	REAL	(e**x)-1
r_floor(x)	REAL	floor
r_hypot(x, y)	REAL	hypotenuse
r_infinity( )	REAL	--

TABLE 1-2 Single-Precision Math Functions *(Continued)*

Function Name	Return Type	Description
r_j0(x)	REAL	Bessel
r_j1(x)	REAL	--
r_jn(x)	REAL	--
ir_finite(x)	INTEGER	--
ir_fp_class(x)	INTEGER	--
ir_ilogb(x)	INTEGER	--
ir_irint(x)	INTEGER	--
ir_isinf(x)	INTEGER	--
ir_isnan(x)	INTEGER	--
ir_isnormal(x)	INTEGER	--
ir_issubnormal(x)	INTEGER	--
ir_iszero(x)	INTEGER	--
ir_signbit(x)	INTEGER	--
r_addran()	REAL	random number generators
r_addrans(x, p, l, u)	subroutineREAL	
r_lcran()		
r_lcrans(x, p, l, u)	subroutine	
r_shufrans(x, p, l, u)	subroutine	
r_lgamma(x)	REAL	log gamma
r_logb(x)	REAL	--
r_log1p(x)	REAL	--
r_log2(x)	REAL	--
r_max_normal()	REAL	
r_max_subnormal()	REAL	
r_min_normal()	REAL	
r_min_subnormal()	REAL	
r_nextafter(x, y)	REAL	
r_quiet_nan(n)	REAL	
r_remainder(x, y)	REAL	
r_rint(x)	REAL	
r_scalb(x, y)	REAL	
r_scalbn(x, n)	REAL	
r_signaling_nan(n)	REAL	
r_significand(x)	REAL	

TABLE 1-2 Single-Precision Math Functions *(Continued)*

Function Name	Return Type	Description
r_sin(x)	REAL	sine
r_sind(x)	REAL	--
r_sinh(x)	REAL	hyperb sin
r_sinp(x)	REAL	--
r_sinpi(x)	REAL	--
r_sincos(x, s, c)	subroutine	sine & cosine
r_sincosd(x, s, c)	subroutine	--
r_sincosp(x, s, c)	subroutine	--
r_sincospi(x, s, c)	subroutine	--
r_tan(x)	REAL	tangent
r_tand(x)	REAL	--
r_tanh(x)	REAL	hyperb tan
r_tanp(x)	REAL	--
r_tanpi(x)	REAL	--
r_y0(x)	REAL	bessel
r_y1(x)	REAL	--
r_yn(n, x)	REAL	--

- 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*.

1.3.2 Double-Precision Functions

The following subprograms are double-precision math 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.

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 1-3 Double Precision Math Functions

Function Name	Return Type	Description
d_acos(x)	DOUBLE PRECISION	arc cosine
d_acosd(x)	DOUBLE PRECISION	--
d_acosh(x)	DOUBLE PRECISION	arc cosh
d_acosp(x)	DOUBLE PRECISION	--
d_acospi(x)	DOUBLE PRECISION	--
d_atan(x)	DOUBLE PRECISION	arc tangent
d_atand(x)	DOUBLE PRECISION	--
d_atanh(x)	DOUBLE PRECISION	arc tanh
d_atanp(x)	DOUBLE PRECISION	--
d_atanpi(x)	DOUBLE PRECISION	--
d_asin(x)	DOUBLE PRECISION	arc sine
d_asind(x)	DOUBLE PRECISION	--
d_asinh(x)	DOUBLE PRECISION	arc sinh
d_asinp(x)	DOUBLE PRECISION	--
d_asinpi(x)	DOUBLE PRECISION	--
d_atan2((y, x)	DOUBLE PRECISION	arc tangent
d_atan2d(y, x)	DOUBLE PRECISION	--
d_atan2pi(y, x)	DOUBLE PRECISION	--
d_cbrt(x)	DOUBLE PRECISION	cube root
d_ceil(x)	DOUBLE PRECISION	ceiling
d_copysign(x, x)	DOUBLE PRECISION	--
d_cos(x)	DOUBLE PRECISION	cosine
d_cosd(x)	DOUBLE PRECISION	--
d_cosh(x)	DOUBLE PRECISION	hyperb cos
d_cosp(x)	DOUBLE PRECISION	--
d_cospi(x)	DOUBLE PRECISION	--
d_erf(x)	DOUBLE PRECISION	error func
d_erfc(x)	DOUBLE PRECISION	--
d_expm1(x)	DOUBLE PRECISION	(e**x)-1
d_floor(x)	DOUBLE PRECISION	floor
d_hypot(x, y)	DOUBLE PRECISION	hypotenuse
d_infinity( )	DOUBLE PRECISION	--

TABLE 1-3 Double Precision Math Functions *(Continued)*

Function Name	Return Type	Description
d_j0(x)	DOUBLE PRECISION	Bessel
d_j1(x)	DOUBLE PRECISION	--
d_jn(x)	DOUBLE PRECISION	--
id_finite(x)	INTEGER	
id_fp_class(x)	INTEGER	
id_ilogb(x)	INTEGER	
id_irint(x)	INTEGER	
id_isinf(x)	INTEGER	
id_isnan(x)	INTEGER	
id_isnormal(x)	INTEGER	
id_issubnormal(x)	INTEGER	
id_iszero(x)	INTEGER	
id_signbit(x)	INTEGER	
d_addran()	DOUBLE PRECISION	random number generators
d_addrans(x, p, l, u)	subroutine	
d_lcran()	DOUBLE PRECISION	
d_lcrans(x, p, l, u)	subroutine	
d_shufrans(x, p, l,u)	subroutine	
d_lgamma(x)	DOUBLE PRECISION	log gamma
d_logb(x)	DOUBLE PRECISION	--
d_log1p(x)	DOUBLE PRECISION	--
d_log2(x)	DOUBLE PRECISION	--
d_max_normal()	DOUBLE PRECISION	
d_max_subnormal()	DOUBLE PRECISION	
d_min_normal()	DOUBLE PRECISION	
d_min_subnormal()	DOUBLE PRECISION	
d_nextafter(x, y)	DOUBLE PRECISION	
d_quiet_nan(n)	DOUBLE PRECISION	
d_remainder(x, y)	DOUBLE PRECISION	
d_rint(x)	DOUBLE PRECISION	
d_scalb(x, y)	DOUBLE PRECISION	
d_scalbn(x, n)	DOUBLE PRECISION	
d_signaling_nan(n)	DOUBLE PRECISION	
d_significand(x)	DOUBLE PRECISION	

TABLE 1-3 Double Precision Math Functions *(Continued)*

Function Name	Return Type	Description
d_sin(x)	DOUBLE PRECISION	sine
d_sind(x)	DOUBLE PRECISION	--
d_sinh(x)	DOUBLE PRECISION	hyper sine
d_sinp(x)	DOUBLE PRECISION	--
d_sinpi(x)	DOUBLE PRECISION	--
d_sincos(x, s, c)	subroutine	sine & cosine
d_sincosd(x, s, c)	subroutine	--
d_sincosp(x, s, c)	subroutine	--
d_sincospi(x, s, c)	subroutine	
d_tan(x)	DOUBLE PRECISION	tangent
d_tand(x)	DOUBLE PRECISION	--
d_tanh(x)	DOUBLE PRECISION	hyperb tan
d_tanp(x)	DOUBLE PRECISION	--
d_tanpi(x)	DOUBLE PRECISION	--
d_y0(x)	DOUBLE PRECISION	bessel
d_y1(x)	DOUBLE PRECISION	--
d_yn(n, x)	DOUBLE PRECISION	--

- 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*.

1.3.3 Quad-Precision Functions

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

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

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

TABLE 1-4 Quadruple-Precision `libm` Functions

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

- 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*.

1.4 Fortran Library Routines Reference

This section details the subroutines and functions in the Fortran library that are part of the Sun Studio Fortran 95 software but are not standard Fortran 95 intrinsics.

A synopsis of the calling interface is presented in a table of the form

data declarations *calling prototype synopsis with arguments*			
argument 1 name	*data type*	*input/output*	*description*
argument 2 name	*data type*	*input/output*	*description*
Return value	*data type*	Output	*description*

Additional man pages are available in section 3f of the Sun Studio man pages. For example, the command `man -s 3f` access will display the man page for the `access()` function. References to man pages appear in this manual as *manpagename*(*section*). For example, a reference to the man page for the `access()` function appears as `access`(3f), and the man page for the Fortran 95 compiler as `f95`(1)

1.4.1 `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.

1.4.2 access: Check File Permissions or Existence

The function is called by:

INTEGER*4 access 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, x, in any order or combination, or blank, where r, w, x have the following meanings:

'r'	Test for read permission
'w'	Test for write permission
'x'	Test for execute permission
' '	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
```

1.4.3 `alarm`: Call Subroutine After a Specified Time

The function is called by:

`INTEGER*4 alarm` `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.

1.4.4 `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 Section 1.4.36, "`mvbits`: Move a Bit Field" on page 1-59, and Chapters 2 and 3 for other functions that manipulate bit fields.

1.4.4.1 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)',
     1         6x 'not(4)'/4o12.11)
        end
demo% f95 tandornot.f
demo% a.out
    and(7,4)     or(7,4)    xor(7,4)      not(4)
 00000000004 00000000007 00000000003 37777777773
demo%
```

Example: lshift, rshift:

```
demo% cat tlrshift.f
       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% f95 tlrshift.f
demo% a.out
 lshift(7,1) rshift(4,1)
 00000000016 00000000002
demo%
```

1.4.4.2 Usage: bic, bis, bit, setbit

For the subroutines and functions

```
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
```

1.4.5 chdir: Change Default Directory

The function is called by:

INTEGER*4 chdir *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.

1.4.6 chmod: Change the Mode of a File

The function is called by:

INTEGER*4 chmod *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.

1.4.7 `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% f95 dat1.f
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-Jan-02
demo%
```

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

1.4.7.1 `date_and_time`: Get Date and Time

This is a Fortran 95 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 February 16, 2000, it generated the following output:

```
date_time    array values:
year= 2000
month_of_year= 2
day_of_month= 16
time difference in minutes= -420
hour of day= 11
minutes of hour= 49
seconds of minute= 29
milliseconds of second= 236
DATE=20000216
TIME=114929.236
ZONE=-0700
```

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

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

Versions of `dtime` and `etime` used by Fortran 95 use the system's low resolution clock by default. The resolution is one hundreth of a second. However, if the program is run under the Sun OS™ operating system utility `ptime`(1), (`/usr/proc/bin/ptime`), the high resolution clock is used.

1.4.8.1 `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 – Calling `dtime` from within a parallelized loop gives non-deterministic results, since the elapsed time counter is global to all threads participating in the loop

---

The function is called by:

e = dtime(*tarray*)			
tarray	real(2)	Output	*e*= -1.0: Error: *tarray* values are undefined *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 *e*≠ -1.0: The sum of *tarray(1)* and *tarray(2)*

Example: `dtime()`, single processor:

```
demo% cat tdtime.f
       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% f95 tdtime.f
demo% a.out
elapsed: 0.0E+0 , user: 0.0E+0 , sys: 0.0E+0
 elapsed: 0.03 , user: 0.01 , sys: 0.02
demo%
```

1.4.8.2 `etime`: Elapsed Time Since Start of Execution

For `etime`, the elapsed time is:

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

The runtime library determines that a program is executing in a multiprocessor mode if either the `PARALLEL` or `OMP_NUM_THREADS` environment variables are defined to some integer value greater than 1.

The function is called by:

e = etime(*tarray*)			
tarray	real(2)	Output	*e*= -1.0: Error: *tarray* values are undefined. *e*≠ -1.0: Single Processor: User time in *tarray(1)*. System time in *tarray(2)* Multiple Processor: Wall clock time in *tarray(1)*, 0.0 in *tarray(2)*
Return value	real	Output	*e*= -1.0: Error *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:

```
demo% cat tetime.f
       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% f95 tetime.f
demo% a.out
elapsed: 0.02 , user: 0.01 , sys: 0.01
demo%
```

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

1.4.9 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).

1.4.10 `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` *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).

1.4.11 `flush`: Flush Output to a Logical Unit

The function is called by:

`INTEGER*4 flush` *n* = `flush(` *lunit* `)`			
lunit	`INTEGER*4`	Input	Logical unit
Return value	`INTEGER*4`	Output	*n* = 0 no error *n* > 0 error number

The `flush` function 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. The function returns a positive error number if an error was encountered; zero otherwise.

See also `fclose`(3S).

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

The function is called by:

`INTEGER*4 fork` *n* = `fork()`			
Return value	`INTEGER*4`	Output	*n*>0: *n*=Process ID of copy *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).

1.4.13 `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.

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

The function is called by:

`INTEGER*4 fseek` *n* = `fseek(` *lunit*, *offset*, *from* `)`			
lunit	`INTEGER*4`	Input	Open logical unit

offset	`INTEGER*4`
	or
	`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 or 8, 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 1=Current position 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 (for example, `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
```

1.4.13.2 `ftell`: Return Current Position of File

The function is called by:

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

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).

1.4.14 `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. (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.

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

The function is called by:

`INTEGER fseeko64` *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 1=Current position 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 (for example, `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
```

1.4.14.2 `ftello64`: Return Current Position of File

The function is called by:

`INTEGER*8 ftello64` *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 *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'
...
```

1.4.15 `getarg`, `iargc`: Get Command-Line Arguments

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

1.4.15.1 `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

1.4.15.2 `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% f95 yarg.f
demo% a.out *.f
1 first.f
2 yarg.f
```

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

1.4.16 `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.

1.4.16.1 `getc`: Get Next Character From `stdin`

The function is called by:

`INTEGER*4 getc`
status `= getc(` *char* `)`

char	`character`	Output	Next character
Return value	`INTEGER*4`	Output	*status*=0: OK *status*=-1: End of file *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
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
^D                  terminated by a CONTROL-D.
-1 377          Next attempt to read returns CONTROL-D
demo%
```

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

1.4.16.2 `fgetc`: Get Next Character From Specified Logical Unit

The function is called by:

`INTEGER*4 fgetc`
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 *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).

1.4.17 getcwd: Get Path of Current Working Directory

The function is called by:

INTEGER*4 getcwd *status* = getcwd(*dirname*)			
dirname	character*n	Output 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 *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>.

1.4.18 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.

If *evalue* is too short to hold the complete string value, the string is truncated to fit in *evalue*.

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).

1.4.19 getfd: Get File Descriptor for External Unit Number

The function is called by:

INTEGER*4 getfd
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).

1.4.20 `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:

```
demo% cat 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`:

```
demo% cat 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% cc -c tgetfilepC.c
demo% f95 tgetfilepC.o tgetfilepF.f
demo% a.out
What is the digit? 3
The digit read by C is 3
demo%
```

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

1.4.21 `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).

1.4.22 `getpid`: Get Process ID

The function is called by:

`INTEGER*4 getpid` *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).

1.4.23 `getuid`, `getgid`: Get User or Group ID of Process

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

1.4.23.1 `getuid`: Get User ID of the Process

The function is called by:

`INTEGER*4 getuid` *uid* = `getuid()`			
Return value	`INTEGER*4`	Output	User ID of the process

1.4.23.2 `getgid`: Get Group ID of the Process

The function is called by:

`INTEGER*4 getgid` `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).

1.4.24 `hostnm`: Get Name of Current Host

The function is called by:

`INTEGER*4 hostnm` *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 *status*>0: Error

Example: `hostnm()`:

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

See also `gethostname`(2).

1.4.25 `idate`: Return Current Date

`idate` 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	Three-element array: 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% f95 tidate.f
demo% a.out
 The date is: 10 8 1998
demo%
```

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

These subprograms provide modes and status required to fully exploit ANSI/IEEE Standard 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 1-5 IEEE Arithmetic Support Routines

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

TABLE 1-5 IEEE Arithmetic Support Routines *(Continued)*

exception	`character`	Input
hdl	`sigfpe_handler_type`	Input
out	`character`	Output
Return value	`INTEGER*4`	Output

See the *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 1-6 `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 = `'exception'`	*in* = `'inexact'` *or* `'division'` *or* `'underflow'` *or* `'overflow'` *or* `'invalid'` *or* `'all'` *or* `'common'`

TABLE 1-6 `ieee_flags`(*action,mode,in,out*) Parameters and Actions *(Continued)*

action = `'set'` set floating-point *mode, in* *out* is unused; returns 0	*mode* = `'direction'`	*in* = `'nearest'` *or* `'tozero'` *or* `'positive'` *or* `'negative'`
	mode = `'exception'`	*in* = `'inexact'` *or* `'division'` *or* `'underflow'` *or* `'overflow'` *or* `'invalid'` *or* `'all'` *or* `'common'`
action = `'get'` test *mode* settings *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 =`'exception'`	*out* = `'inexact'` *or* `'division'` *or* `'underflow'` *or* `'overflow'` *or* `'invalid'` *or* `'all'` *or* `'common'`

TABLE 1-7 `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 `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:

```
      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
```

Change the declarations for `address` and `function hand` to `INTEGER*8` to enable Example 5 in a 64-bit, SPARC V9 environment (`-xarch=v9`)

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

1.4.26.1 `floatingpoint.h`: Fortran IEEE Definitions

The header file `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 95 source program as follows:

```
#include "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`, `.F90` or `.F95` extension.

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`(3F).

1.4.27 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:

1.4.27.1 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* *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).

1.4.27.2 rindex: Last Occurrence of a Substring in a String

The function is called by:

INTEGER*4 rindex *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* *n*=0: *a2* does not occur in *a1* INTEGER*8 returned in 64-bit environments

1.4.27.3 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* *n*=0: *a1* is all nonblank INTEGER*8 returned in 64-bit environments

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

```
demo% cat tindex.f
*                          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% f95 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.

1.4.28 inmax: Return Maximum Positive Integer

The function is called by:

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

Example: `inmax`:

```
demo% cat tinmax.f
       INTEGER*4 inmax, m
       m = inmax()
       write(*,*) m
       end
demo% f95 tinmax.f
demo% a.out
   2147483647
demo%
```

See also `libm_single`(3F) and `libm_double`(3F). See also the non-standard FORTRAN 77 intrinsic function `ephuge()` described in Chapter 3.

1.4.29 `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: *iarray*(1) = hour *iarray*(2) = minute *iarray*(3) = second

Example: `itime`:

```
demo% cat titime.f
       INTEGER*4 iarray(3)
       call itime( iarray )
       write (*, "(' The time is: ',3i5)" )  iarray
       end
demo% f95 titime.f
demo% a.out
 The time is: 15 42 35
```

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

1.4.30 `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 *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).

1.4.31 `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
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*. *name2* must not already exist.
Return value	`INTEGER*4`	Output	*status*=0: OK *status*>0: System error code

1.4.31.1 `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% f95 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%
```

1.4.31.2 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% f95 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>.

1.4.32 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.

1.4.33 `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.

1.4.33.1 `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

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

The function is:

`INTEGER*2 short` `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.

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

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

1.4.34.1 `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 *ival* ≠ 0 if `isetjmp` is called through `longjmp`

1.4.34.2 `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 *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 `f95` 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).

1.4.35 `malloc`, `malloc64`, `realloc`, `free`: Allocate/Reallocate/Deallocate Memory

The functions `malloc()`, `malloc64()`, and `realloc()` allocate blocks of memory and return the starting address of the block. The return value can be used to set an `INTEGER` or Cray-style `POINTER` variable. `realloc()` reallocates an existing memory block with a new size. `free()` deallocates memory blocks allocated by `malloc()`, `malloc64()`, or `realloc()`.

Note – These routines are implemented as intrinsic functions in `f95`, but were external functions in `f77`. They should not appear on type declarations in Fortran 95 programs, or on `EXTERNAL` statements unless you wish to use your own versions. The `realloc()` routine is only implemented for `f95`.

Standard-conforming Fortran 95 programs should use `ALLOCATE` and `DEALLOCATE` statements on `ALLOCATABLE` arrays to perform dynamic memory management, and not make direct calls to `malloc/realloc/free`.

Legacy Fortran 77 programs could use `malloc()/malloc64()` to assign values to Cray-style `POINTER` variables, which have the same data representation as `INTEGER` variables. Cray-style `POINTER` variables are implemented in `f95` to support portability from Fortran 77.

1.4.35.1 Allocate Memory: `malloc`, `malloc64`

The `malloc()` function is called by:

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

Note – This function is intrinsic in Fortran 95 and was external in Fortran 77. Fortran 77 programs compiled to run in 64-bit environments would declare the `malloc()` function and the variables receiving its output as `INTEGER*8`. The function `malloc64`(3F) was 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` *(Cray* `POINTER`*)*	Output	*k*>0: *k*=address of *the* start of the block of memory allocated *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)
      integer ( p2X, X )
      real*4 X(1)
      …
      p2X = malloc( NX*4 )
      if ( p2X .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 p2X, and initialize it to zero.

1.4.35.2 Reallocate Memory: realloc

The realloc() f95 intrinsic function is called by:

k = realloc(*ptr*, *n*)			
ptr	INTEGER	Input	Pointer to existing memory block. (Value returned from a previous malloc() or realloc() call).
n	INTEGER	Input	Requested new size of block, in bytes.
Return value	INTEGER *(Cray* POINTER*)*	Output	*k*>0: *k*=address of *the* start of the new block of memory allocated *k*=0: Error
	An INTEGER*8 pointer value is returned when compiled for a 64-bit environment with -xarch=v9. See Note below.		

The realloc() function changes the size of the memory block pointed to by ptr to n bytes and returns a pointer to the (possibly moved) new block. The contents of the memory block will be unchanged up to the lesser of the new and old sizes.

If ptr is zero, realloc() behaves the same as malloc() and allocates a new memory block of size n bytes.

If n is zero and ptr is not zero, the memory block pointed to is made available for further allocation and is returned to the system only upon termination of the application.

Example: Using malloc() and realloc() and Cray-style POINTER variables:

```
PARAMETER (nsize=100001)
POINTER (p2space,space)
REAL*4 space(1)

p2space = malloc(4*nsize)
if(p2space .eq. 0) STOP 'malloc: cannot allocate space'
...
p2space = realloc(p2space, 9*4*nsize)
if(p2space .eq. 0) STOP 'realloc: cannot reallocate space'
...
CALL free(p2space)
...
```

Note that realloc() is only implemented for f95.

1.4.35.3 free: Deallocate Memory Allocated by Malloc

The subroutine is called by:

call free (*ptr*)		
ptr	*Cray* POINTER	Input

free deallocates a region of memory previously allocated by malloc and realloc(). 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
```

1.4.36 `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
     1    / 7,    0,     3,   1,    3 /
       call mvbits ( src, ini1, nbits, des, ini2 )
       write (*,"(5o3)") src, ini1, nbits, des, ini2
       end
demo% f95 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

1.4.37 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.

1.4.37.1 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" )
```

1.4.37.2 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
```

1.4.37.3 `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 `f95` are listed in the *Fortran User's Guide*.

1.4.38 `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.

Note that to write any of the special \ escape characters, such as `'\n'` newline, requires compiling with `-f77=backslash` FORTRAN 77 compatibility option.

1.4.38.1 putc: Write to Logical Unit 6

The function is called by:

```
INTEGER*4 putc
status = putc( char )
```

char	character	Input	The character to write to the unit
Return value	INTEGER*4	Output	*status*=0: OK *status*>0: System error code

Example: `putc()`:

```
demo% cat tputc.f
       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% f95 -f77=backslash tputc.f
demo% a.out
OK by putc
demo%
```

1.4.38.2 fputc: Write to Specified Logical Unit

The function is called by:

```
INTEGER*4 fputc
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 *status*>0: System error code

Example: `fputc()`:

```
demo% cat tfputc.f
       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% f95 -f77=backslash tfputc.f
demo% a.out
demo% cat tfputc.data
OK by fputc
demo%
```

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

1.4.39 `qsort,qsort64`: Sort the Elements of a One-Dimensional Array

The subroutine is called by:

`call qsort( array, len, isize, compar )` *(Continued)*
`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: 4 for integer or real 8 for double precision or complex 16 for double complex Length of character object for character arrays
isize8	INTEGER*8	Input	Size of an element, typically: 4_8 for integer or real 8_8 for double precision or complex 16_8 for double complex Length of character object for character arrays
compar	function name	Input	Name of a user-supplied INTEGER*2 function. 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 95 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/,
     1           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% f95 tqsort.f
demo% a.out
  0 1 2 3 4 5 6 7 8 9
```

1.4.40 ran: Generate a Random Number Between 0 and 1

Repeated calls to ran generate a sequence of random numbers with a uniform distribution. See lcrans(3m).

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

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% f95 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.

1.4.41 `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*)
r = `rand`(*k*)
d = `drand`(*k*)

k	`INTEGER*4`	Input	*k*=0: Get next random number in the sequence *k*=1: Restart sequence, return first number *k*>0: Use as a seed for new sequence, return first number
`rand`	`REAL*4`	Output	
`drand`	`REAL*8`	Output	
`irand`	`INTEGER*4`	Output	

Example: `irand()`:

```
demo% cat trand.f
       integer*4 v(5), iflag/0/
       do i = 1, 5
        v(i) = irand( iflag )
       end do
       write(*,*) v
       end
demo% f95 trand.f
demo% a.out
   2078917053 143302914 1027100827 1953210302 755253631
demo%
```

1.4.42 `rename`: Rename a File

The function is called by:

`INTEGER*4 rename`
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 *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% f95 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>`.

1.4.43 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, 10000
        x = asin( y )
       end do
       elapsed = secnds( t1 )
       write ( *, 1 ) elapsed
  1    format ( ' 10000 arcsines: ', f12.6, ' sec' )
       end
demo% f95 sec1.f
demo% a.out
 10000 arcsines:     0.009064 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.

1.4.44 set_io_err_handler, get_io_err_handler: Set and Get I/O Error Handler

set_io_err_handler() declares a user-defined routine to be called whenever errors are detected on a specified input logical unit.

`get_io_err_handler()` returns the address of the currently declared error handling routine.

These routines are module subroutines and can only be accessed when `USE SUN_IO_HANDLERS` appears in the calling routine.

`USE SUN_IO_HANDLERS`
`call set_io_err_handler(`*iu, subr_name, istat*`)`

iu	`INTEGER*8`	Input	Logical unit number
subr_name	`EXTERNAL`	Input	Name of user-supplied error handler subroutine.
istat	`INTEGER*4`	Output	Return status.

`USE SUN_IO_HANDLERS`
`call get_io_err_handler(`*iu, subr_pointer, istat*`)`

iu	`INTEGER*8`	Input	Logical unit number
subr_pointer	`POINTER`	Output	Address of currenly declared handler routine.
istat	`INTEGER*4`	Output	Return status.

`SET_IO_ERR_HANDLER` sets the user-supplied subroutine *subr_name* to be used as the I/O error handler for the logical unit *iu* when an input error occurs. *iu* has to be a connected Fortran logical unit for a formatted file. *istat* will be set to a non-zero value if there is an error, otherwise it is set to zero.

For example, if `SET_IO_ERR_HANDLER` is called before the logical unit *iu* has been opened, *istat* will be set to 1001 ("Illegal Unit"). If *subr_name* is NULL user error-handling is turned off and the program reverts to default Fortran error handling.

Use `GET_IO_ERR_HANDLER` to get the address of the function currently being used as the error handler for this logical unit. For example, call `GET_IO_ERR_HANDLER` to save the current I/O before switching to another handler routine. The error handler can be restored with the saved value later.

subr_name is the name of the user-supplied routine to handle the I/O error on logical unit *iu*. The runtime I/O library passes all relevant information to *subr_name*, enabling this routine to diagnose the problem and possibly fix the error before continuing.

The interface to the user-supplied error handler routine is as follows:

```
SUBROUTINE SUB_NAME(UNIT, SRC_FILE, SRC_LINE, DATA_FILE, FILE_POS,
                    CURR_BUFF, CURR_ITEM, CORR_CHAR, CORR_ACTION )
INTENT (IN) UNIT, SRC_FILE, SRC_LINE, DATA_FILE
INTENT (IN) FILE_POS, CURR_BUFF, CURR_ITEM
INTENT (OUT) CORR_CHAR, CORR_ACTION
```

UNIT	INTEGER*8	Input	Logical unit number of the input file reporting an error.
SRC_FILE	CHARACTER*(*)	Input	Name of the Fortran source file originating the input operation.
SRC_LINE	INTEGER*8	Input	Line number in SRC_FILE of the input operation with an error.
DATA_FILE	CHARACTER*(*)	Input	Name of the data file being read. Avaliable only if the file is an opened external file. If the name is not available, (logical unit 5 for example), DATA_FILE is set to a zero-length character data item.
FILE_POS	INTEGER*8	Input	Current position in the input file, in bytes. Defined only if the name of the DATA_FILE is known.
CURR_BUFF	CHARACTER*(*)	Input	Character string containing the remaining data from the input record. The bad input character is the first character in the string.

CURR_ITEM	INTEGER*8	Input	The number of input items in the record that have been read, including the current one, when the error is detected. For example: READ(12,10)L,(ARR(I),I=1,L) IF the value of CURR_ITEM is 15 in this case, it means the error happens while reading the 14th element of ARR, with L being the first item and ARR(1) being the second, and so on.
CORR_CHAR	CHARACTER	Output	The user-supplied corrected character to be returned by the handler. This value is used only when CORR_ACTION is non-zero. If CORR_CHAR is an invalid character, the handler will be called again until a valid character is returned. This could cause an infinite loop, a situation the user is required to protect against.
CORR_ACTION	INTEGER	Output	Specifies the corrective action to be taken by the I/O library. With a zero value no special action is taken and the library reverts to its default error processing. A value of 1 returns CORR_CHAR to the I/O error processing routine.

Limitations

The I/O handler can only replace once character with another character. It cannot replace one character with more than one character.

The error recovery algorithm can only fix a bad character it currently reads, not a bad character which has already been interpreted as a valid character in a different context. For example,in a list-directed read, if the input is "1.234509.8765" when the correct input should be "1.2345 9.8765" then the I/O library will run into an error at the second period because it is not a valid number. But, by that time, it is not possible to go back and change the '0' into a blank.

Currently, this error-handling capability does not work for namelist-directed input. When doing namelist-directed input, the specified I/O error handler will not be invoked when an error occurs.

I/O error handlers can only be set for external files, not internal files, because there are no logical units associated with internal files.

The I/O error handler is called only for syntactic errors, not system errors or semantic errors(such as an overflowed input value).

It is possible to have an infinite loop if the user-supplied I/O error handler keeps providing a bad character to the I/O library, causing it to call the user-supplied I/O error handler over and over again. If an error keeps occuring at the same file position then the error handler should terminate itself. One way to do this is to take the default error path by setting `CORR_ACTION` to 0. Then the I/O library will continue with the normal error handling.

1.4.45 `sh`: Fast Execution of an `sh` Command

The function is called by:

`INTEGER*4 sh`
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.

1.4.46 `signal`: Change the Action for a Signal

The function is called by:

`INTEGER*4 signal` *or* `INTEGER*8 signal` *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 *flag* ≥ 0: Ignore *proc*; pass *flag* as the action: *flag* = 0: Use the default action *flag* = 1: Ignore this signal
Return value	`INTEGER*4`	Output	*n*=-1: System error *n*>0: Definition of previous action *n*>1: *n*=Address of routine that would have been called *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.
	`INTEGER*8`		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.

`floatingpoint.h` defines *proc* values `SIGFPE_DEFAULT`, `SIGFPE_IGNORE`, and `SIGFPE_ABORT`. See Section 1.4.26.1, "`floatingpoint.h`: Fortran IEEE Definitions" on page 1-44.

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*.

1.4.47 `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).

1.4.48 `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.

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

The function is called by:

`INTEGER*4 stat`
ierr `= stat (` *name*, *statb* `)`

name	`character*`*n*	Input	Name of the file
statb	`INTEGER*4`	Output	Status structure for the file, 13-element array
Return value	`INTEGER*4`	Output	*ierr*=0: OK *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),',
     1   blocks = ',statb(13)
      end
```

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

The function

`INTEGER*4 fstat` *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 *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),',
     1      blocks = ',statb(13)
      end
```

1.4.48.3 `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 *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),',
     1   blocks = ',statb(13)
      end
```

1.4.48.4 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>`.

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

64-bit "long file" 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`.

1.4.50 `system`: Execute a System Command

The function is called by:

`INTEGER*4 system` *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` developed with different assumptions:

- If `cc` calls `system`, the shell is always the Bourne 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.

1.4.51 `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.

1.4.51.1 `time`: Get System Time

The *time()* function is called by:

`INTEGER*4 time` *or* `INTEGER*8` *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:

```
demo% cat ttime.f
        INTEGER*4  n, time
        n = time()
        write(*,*) 'Seconds since 0 1/1/70 GMT = ', n
        end
demo% f95 ttime.f
demo% a.out
 Seconds since 0 1/1/70 GMT =   913240205
demo%
```

1.4.51.2 `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` *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()`:

```
demo% cat tctime.f
        character*24 ctime, string
        INTEGER*4  n, time
        n = time()
        string = ctime( n )
        write(*,*) 'ctime: ', string
        end
demo% f95 tctime.f
demo% a.out
 ctime: Wed Dec  9 13:50:05 1998
demo%
```

1.4.51.3 `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()`:

```
demo% cat tltime.f
        integer*4  stime, tarray(9), time
        stime = time()
        call ltime( stime, tarray )
        write(*,*) 'ltime: ', tarray
        end
demo% f95 tltime.f
demo% a.out
 ltime: 25 49 10 12 7 91 1 223 1
demo%
```

1.4.51.4 `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`:

```
demo% cat tgmtime.f
        integer*4  stime, tarray(9), time
        stime = time()
        call gmtime( stime, tarray )
        write(*,*) 'gmtime: ', tarray
        end
demo% f95t 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).

1.4.51.5 `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.

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

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

1.4.52.1 `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 *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. If empty string (all blanks) returned: *lunit* is not associated with a terminal device in the directory, /dev

1.4.52.2 isatty: Is this Unit a Terminal?

The function isatty returns true or false depending on whether or not logical unit *lunit* is a terminal device or not.

The function is called by:

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

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   "
```

1.4.53 `unlink`: Remove a File

The function is called by:

`INTEGER*4 unlink` *n* = unlink (*patnam*)			
patnam	`character*`*n*	Input	File name
Return value	`INTEGER*4`	Output	*n*=0: OK *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:

```
demo% cat tunlink.f
       call unlink( 'tunlink.data' )
       end
demo% f95 tunlink.f
demo% ls tunl*
tunlink.f tunlink.data
demo% a.out
demo% ls tunl*
tunlink.f
```

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

1.4.54 `wait`: Wait for a Process to Terminate

The function is:

`INTEGER*4 wait` *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 *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).

CHAPTER 2

Fortran 95 Intrinsic Functions

This chapter lists the intrinsic function names recognized by the `f95` compiler.

2.1 Standard Fortran 95 Generic Intrinsic Functions

The generic Fortran 95 intrinsic functions are grouped in this section by functionality as they appear in the Fortran 95 standard.

The arguments shown are the names that can be used as argument keywords when using the keyword form, as in `cmplx(Y=B, KIND=M, X=A)`.

Consult the Fortran 95 standard for the detailed specifications of these generic intrinsic procedures.

2.1.1 Argument Presence Inquiry Function

Generic Intrinsic Name	Description
PRESENCE	Argument presence

2.1.2 Numeric Functions

Generic Intrinsic Name	Description
`ABS (A)`	Absolute value
`AIMAG (Z)`	Imaginary part of a complex number
`AINT (A [, KIND])`	Truncation to whole number
`ANINT (A [, KIND])`	Nearest whole number
`CEILING (A [, KIND])`	Least integer greater than or equal to number
`CMPLX (X [, Y, KIND])`	Conversion to complex type
`CONJG (Z)`	Conjugate of a complex number
`DBLE (A)`	Conversion to double precision real type
`DIM (X, Y)`	Positive difference
`DPROD (X, Y)`	Double precision real product
`FLOOR (A [, KIND])`	Greatest integer less than or equal to number
`INT (A [, KIND])`	Conversion to integer type
`MAX (A1, A2 [, A3,...])`	Maximum value
`MIN (A1, A2 [, A3,...])`	Minimum value
`MOD (A, P)`	Remainder function
`MODULO (A, P)`	Modulo function
`NINT (A [, KIND])`	Nearest integer
`REAL (A [, KIND])`	Conversion to real type
`SIGN (A, B)`	Transfer of sign

2.1.3 Mathematical Functions

Generic Intrinsic Name	Description
`ACOS (X)`	Arccosine
`ASIN (X)`	Arcsine
`ATAN (X)`	Arctangent
`ATAN2 (Y, X)`	Arctangent
`COS (X)`	Cosine

Generic Intrinsic Name	Description
COSH (X)	Hyperbolic cosine
EXP (X)	Exponential
LOG (X)	Natural logarithm
LOG10 (X)	Common logarithm (base 10)
SIN (X)	Sine
SINH (X)	Hyperbolic sine
SQRT (X)	Square root
TAN (X)	Tangent
TANH (X)	Hyperbolic tangent

2.1.4 Character Functions

Generic Intrinsic Name	Description
ACHAR (I)	Character in given position in ASCII collating sequence
ADJUSTL (STRING)	Adjust left
ADJUSTR (STRING)	Adjust right
CHAR (I [, KIND])	Character in given position in processor collating sequence
IACHAR (C)	Position of a character in ASCII collating sequence
ICHAR (C)	Position of a character in processor collating sequence
INDEX (STRING, SUBSTRING [, BACK])	Starting position of a substring
LEN_TRIM (STRING)	Length without trailing blank characters
LGE (STRING_A, STRING_B)	Lexically greater than or equal
LGT (STRING_A, STRING_B)	Lexically greater than
LLE (STRING_A, STRING_B)	Lexically less than or equal
LLT (STRING_A, STRING_B)	Lexically less than
REPEAT (STRING, NCOPIES)	Repeated concatenation

Generic Intrinsic Name	Description
SCAN (STRING, SET [, BACK])	Scan a string for a character in a set
TRIM (STRING)	Remove trailing blank characters
VERIFY (STRING, SET [, BACK])	Verify the set of characters in a string

2.1.5 Character Inquiry Function

Generic Intrinsic Name	Description
LEN (STRING)	Length of a character entity

2.1.6 Kind Functions

Generic Intrinsic Name	Description
KIND (X)	Kind type parameter value
SELECTED_INT_KIND (R)	Integer kind type parameter value, given range
SELECTED_REAL_KIND ([P, R])	Real kind type parameter value, given precision and range

2.1.7 Logical Function

Generic Intrinsic Name	Description
LOGICAL (L [, KIND])	Convert between objects of type logical with different kind type parameters

2.1.8 Numeric Inquiry Functions

Generic Intrinsic Name	Description
DIGITS (X)	Number of significant digits of the model
EPSILON (X)	Number that is almost negligible compared to one
HUGE (X)	Largest number of the model
MAXEXPONENT (X)	Maximum exponent of the model
MINEXPONENT (X)	Minimum exponent of the model
PRECISION (X)	Decimal precision
RADIX (X)	Base of the model
RANGE (X)	Decimal exponent range
TINY (X)	Smallest positive number of the model

2.1.9 Bit Inquiry Function

Generic Intrinsic Name	Description
BIT_SIZE (I)	Number of bits of the model

2.1.10 Bit Manipulation Functions

Generic Intrinsic Name	Description
BTEST (I, POS)	Bit testing
IAND (I, J)	Logical AND
IBCLR (I, POS)	Clear bit
IBITS (I, POS, LEN)	Bit extraction
IBSET (I, POS)	Set bit
IEOR (I, J)	Exclusive OR
IOR (I, J)	Inclusive OR

Generic Intrinsic Name	Description
ISHFT (I, SHIFT)	Logical shift
ISHFTC (I, SHIFT [, SIZE])	Circular shift
NOT (I)	Logical complement

2.1.11 Transfer Function

Generic Intrinsic Name	Description
TRANSFER (SOURCE, MOLD [, SIZE])	Treat first argument as if of type of second argument

2.1.12 Floating-Point Manipulation Functions

Generic Intrinsic Name	Description
EXPONENT (X)	Exponent part of a model number
FRACTION (X)	Fractional part of a number
NEAREST (X, S)	Nearest different processor number in given direction
RRSPACING (X)	Reciprocal of the relative spacing of model numbers near given number
SCALE (X, I)	Multiply a real by its base to an integer power
SET_EXPONENT (X, I)	Set exponent part of a number
SPACING (X)	Absolute spacing of model numbers near given number

2.1.13 Vector and Matrix Multiply Functions

Generic Intrinsic Name	Description
DOT_PRODUCT (VECTOR_A, VECTOR_B)	Dot product of two rank-one arrays
MATMUL (MATRIX_A, MATRIX_B)	Matrix multiplication

2.1.14 Array Reduction Functions

Generic Intrinsic Name	Description
`ALL (MASK [, DIM])`	True if all values are true
`ANY (MASK [, DIM])`	True if any value is true
`COUNT (MASK [, DIM])`	Number of true elements in an array
`MAXVAL (ARRAY, DIM [, MASK])` or `MAXVAL (ARRAY [, MASK])`	Maximum value in an array
`MINVAL (ARRAY, DIM [, MASK])` or `MINVAL (ARRAY [, MASK])`	Minimum value in an array
`PRODUCT (ARRAY, DIM [, MASK])` or `PRODUCT (ARRAY [, MASK])`	Product of array elements
`SUM (ARRAY, DIM [, MASK])` or `SUM (ARRAY [, MASK])`	Sum of array elements

2.1.15 Array Inquiry Functions

Generic Intrinsic Name	Description
`ALLOCATED (ARRAY)`	Array allocation status
`LBOUND (ARRAY [, DIM])`	Lower dimension bounds of an array
`SHAPE (SOURCE)`	Shape of an array or scalar
`SIZE (ARRAY [, DIM])`	Total number of elements in an array
`UBOUND (ARRAY [, DIM])`	Upper dimension bounds of an array

2.1.16 Array Construction Functions

Generic Intrinsic Name	Description
`MERGE (TSOURCE, FSOURCE, MASK)`	Merge under mask

Generic Intrinsic Name	Description
PACK (ARRAY, MASK [, VECTOR])	Pack an array into an array of rank one under a mask
SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask

2.1.17 Array Reshape Function

Generic Intrinsic Name	Description
RESHAPE (SOURCE, SHAPE[, PAD, ORDER])	Reshape an array

2.1.18 Array Manipulation Functions

Generic Intrinsic Name	Description
CSHIFT (ARRAY, SHIFT [, DIM])	Circular shift
EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM]	End-off shift
TRANSPOSE (MATRIX)	Transpose of an array of rank two

2.1.19 Array Location Functions

Generic Intrinsic Name	Description
MAXLOC (ARRAY, DIM [, MASK]) or MAXLOC (ARRAY [, MASK])	Location of a maximum value in an array
MINLOC (ARRAY, DIM [, MASK]) or MINLOC (ARRAY [, MASK])	Location of a minimum value in an array

2.1.20 Pointer Association Status Functions

Generic Intrinsic Name	Description
`ASSOCIATED (POINTER [, TARGET])`	Association status inquiry or comparison
`NULL ([MOLD])`	Returns disassociated pointer

2.1.21 System Environment Procedures

Generic Intrinsic Name	Description
`COMMAND_ARGUMENT_COUNT ()`	Returns number of command arguments
`GET_COMMAND ([COMMAND, LENGTH, STATUS])`	Returns entire command that invoked the program
`GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])`	Returns a command argument
`GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])`	Obtain the value of an environment variable.

2.1.22 Intrinsic Subroutines

Generic Intrinsic Name	Description
`CPU_TIME (TIME)`	Obtain processor time
`DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])`	Obtain date and time
`MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)`	Copies bits from one integer to another
`RANDOM_NUMBER (HARVEST)`	Returns pseudorandom number
`RANDOM_SEED ([SIZE, PUT, GET])`	Initializes or restarts the pseudorandom number generator
`SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])`	Obtain data from the system clock

2.1.23 Specific Names for Intrinsic Functions

TABLE 2-1 Specific and Generic Names for Fortran 95 Intrinsic Functions

	Specific Name	Generic Name	Argument Type
	ABS (A)	ABS (A)	default real
	ACOS (X)	ACOS (X)	default real
	AIMAG (Z)	AIMAG (Z)	default complex
	AINT (A)	AINT (A)	default real
	ALOG (X)	LOG (X)	default real
	ALOG10 (X)	LOG10 (X)	default real
#	AMAX0 (A1, A2 [, A3,...])	REAL (MAX (A1, A2 [, A3,...]))	default integer
#	AMAX1 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	default real
#	AMIN0 (A1, A2 [, A3,...])	REAL (MIN (A1, A2 [, A3,...]))	default integer
#	AMIN1 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	default real
	AMOD (A, P)	MOD (A, P)	default real
	ANINT (A)	ANINT (A)	default real
	ASIN (X)	ASIN (X)	default real
	ATAN (X)	ATAN (X)	default real
	ATAN2 (Y, X)	ATAN2 (Y, X)	default real
	CABS (A)	ABS (A)	default complex
	CCOS (X)	COS (X)	default complex
	CEXP (X)	EXP (X)	default complex
#	CHAR (I)	CHAR (I)	default integer
	CLOG (X)	LOG (X)	default complex
	CONJG (Z)	CONJG (Z)	default complex
	COS (X)	COS (X)	default real
	COSH (X)	COSH (X)	default real
	CSIN (X)	SIN (X)	default complex
	CSQRT (X)	SQRT (X)	default complex
	DABS (A)	ABS (A)	double precision
	DACOS (X)	ACOS (X)	double precision
	DASIN (X)	ASIN (X)	double precision

TABLE 2-1 Specific and Generic Names for Fortran 95 Intrinsic Functions *(Continued)*

	Specific Name	Generic Name	Argument Type
	DATAN (X)	ATAN (X)	double precision
	DATAN2 (Y, X)	ATAN2 (Y, X)	double precision
	DCOS (X)	COS (X)	double precision
	DCOSH (X)	COSH (X)	double precision
	DDIM (X, Y)	DIM (X, Y)	double precision
	DEXP (X)	EXP (X)	double precision
	DIM (X, Y)	DIM (X, Y)	default real
	DINT (A)	AINT (A)	double precision
	DLOG (X)	LOG (X)	double precision
	DLOG10 (X)	LOG10 (X)	double precision
#	DMAX1 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	double precision
#	DMIN1 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	double precision
	DMOD (A, P)	MOD (A, P)	double precision
	DNINT (A)	ANINT (A)	double precision
	DPROD (X, Y)	DPROD (X, Y)	default real
	DSIGN (A, B)	SIGN (A, B)	double precision
	DSIN (X)	SIN (X)	double precision
	DSINH (X)	SINH (X)	double precision
	DSQRT (X)	SQRT (X)	double precision
	DTAN (X)	TAN (X)	double precision
	DTANH (X)	TANH (X)	double precision
	EXP (X)	EXP (X)	default real
#	FLOAT (A)	REAL (A)	default integer
	IABS (A)	ABS (A)	default integer
#	ICHAR (C)	ICHAR (C)	default character
	IDIM (X, Y)	DIM (X, Y)	default integer
#	IDINT (A)	INT (A)	double precision
	IDNINT (A)	NINT (A)	double precision
#	IFIX (A)	INT (A)	default real
	INDEX (STRING, SUBSTRING)	INDEX (STRING, SUBSTRING)	default character

TABLE 2-1 Specific and Generic Names for Fortran 95 Intrinsic Functions *(Continued)*

	Specific Name	Generic Name	Argument Type
#	INT (A)	INT (A)	default real
	ISIGN (A, B)	SIGN (A, B)	default integer
	LEN (STRING)	LEN (STRING)	default character
#	LGE (STRING_A, STRING_B)	LGE (STRING_A, STRING_B)	default character
#	LGT (STRING_A, STRING_B)	LGT (STRING_A, STRING_B)	default character
#	LLE (STRING_A, STRING_B)	LLE (STRING_A, STRING_B)	default character
#	LLT (STRING_A, STRING_B)	LLT (STRING_A, STRING_B)	default character
#	MAX0 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	default integer
#	MAX1 (A1, A2 [, A3,...])	INT (MAX (A1, A2 [, A3,...]))	default real
#	MIN0 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	default integer
#	MIN1 (A1, A2 [, A3,...])	INT (MIN (A1, A2 [, A3,...]))	default real
	MOD (A, P)	MOD (A, P)	default integer
	NINT (A)	NINT (A)	default real
#	REAL (A)	REAL (A)	default integer
	SIGN (A, B)	SIGN (A, B)	default real
	SIN (X)	SIN (X)	default real
	SINH (X)	SINH (X)	default real
#	SNGL (A)	REAL (A)	double precision
	SQRT (X)	SQRT (X)	default real
	TAN (X)	TAN (X)	default real
	TANH (X)	TANH (X)	default real

Functions marked with # cannot be used as an actual argument.
"double precision" means double-precision real.

2.2 Fortran 2000 Module Routines

The Fortran 2000 draft standard provides a set of intrinsic modules that define features to support IEEE arithmetic and interoperability with the C language. These modules define new functions and subroutines, and are implemented in the Sun Studio 8 Fortran 95 compiler.

2.2.1 IEEE Arithmetic and Exceptions Modules

The Fortran 2000 draft standard intrinsic modules `IEEE_EXCEPTIONS`, `IEEE_ARITHMETIC`, and `IEEE_FEATURES` to support new features in the proposed language standard to support IEEE arithmetic and IEEE exception handling.

The draft standard defines a set of inquiry functions, elemental functions, kind functions, elemental subroutines, and nonelemental subroutines. These are listed in the tables that follow.

To access these functions and subroutines, the calling routine must include

```
USE, INTRINSIC :: IEEE_ARITHMETIC, IEEE_EXCEPTIONS
```

See Chapter 14 of the draft standard (http://www.j3-fortran.org) for details.

2.2.1.1 Inquiry Functions

The module `IEEE_EXCEPTIONS` contains the following inquiry functions.

Function	Descripton
`IEEE_SUPPORT_FLAG(FLAG[,X])`	Inquire whether the processor supports an exception.
`IEEE_SUPPORT_HALTING(FLAG)`	Inquire whether the processor supports control of halting after an exception.

The module `IEEE_ARITHMETIC` contains the following inquiry functions.

Function	Description
`IEEE_SUPPORT_DATATYPE([X])`	Inquire whether the processor supports IEEE arithmetic.
`IEEE_SUPPORT_DENORMAL([X])`	Inquire whether the processor supports denormalized numbers.
`IEEE_SUPPORT_DIVIDE([X])`	Inquire whether the processor supports divide with the accuracy specified by the IEEE standard.
`IEEE_SUPPORT_INF([X])`	Inquire whether the processor supports the IEEE infinity.
`IEEE_SUPPORT_IO([X])`	Inquire whether the processor supports IEEE base conversion rounding during formatted input/output.
`IEEE_SUPPORT_NAN([X])`	Inquire whether the processor supports the IEEE Not-a-Number.

IEEE_SUPPORT_ROUNDING(VAL[,X])	Inquire whether the processor supports a particular rounding mode.
IEEE_SUPPORT_SQRT([X])	Inquire whether the processor supports the IEEE square root.
IEEE_SUPPORT_STANDARD([X])	Inquire whether the processor supports all IEEE facilities.

2.2.1.2 Elemental Functions

The module IEEE_ARITHMETIC contains the following elemental functions for real X and Y for which IEEE_SUPPORT_DATATYPE(X) and IEEE_SUPPORT_DATATYPE(Y) are true.

Function	Description
IEEE_CLASS(X)	IEEE class.
IEEE_COPY_SIGN(X,Y)	IEEE copysign function.
IEEE_IS_FINITE(X)	Determine if value is finite.
IEEE_IS_NAN(X)	Determine if value is IEEE Not-a-Number
IEEE_IS_NORMAL(X)	Determine if a value is normal.
IEEE_IS_NEGATIVE(X)	Determine if value is negative.
IEEE_LOGB(X)	Unbiased exponent in the IEEE floating point format.
IEEE_NEXT_AFTER(X,Y)	Returns the next representable neighbor of X in the direction toward Y.
IEEE_REM(X,Y)	The IEEE REM remainder function, X - Y*N where N is the integer nearest to the exact value of X/Y.
IEEE_RINT(X)	Round to an integer value according to the current rounding mode.
IEEE_SCALB(X,I)	Returns X*2**I
IEEE_UNORDERED(X,Y)	IEEE unordered function. True if X or Y is a NaN and false otherwise.
IEEE_VALUE(X,CLASS)	Generate an IEEE value.

2.2.1.3 Kind Function

The module IEEE_ARITHMETIC contains the following transformational function:

Function	Description
IEEE_SELECTED_REAL_KIND([P,][R])	Kind type parameter value fo an IEEE real with given precision and range.

2.2.1.4 Elemental Subroutines

The module IEEE_EXCEPTIONS contains the following elemental subroutines.

Subroutine	Description
IEEE_GET_FLAG(FLAG,FLAG_VALUE)	Get an exception flag.
IEEE_GET_HALTING_MODE(FLAG,HALTING)	GEt halting mode for an exception.

2.2.1.5 Nonelemental Subroutines

The module IEEE_EXCEPTIONS contains the following nonelemental subroutines.

Subroutine	Description
IEEE_GET_STATUS(STATUS_VALUE)	Get the current state of the floating point environment.
IEEE_SET_FLAG(FLAG,FLAG_VALUE)	Set an exception flag.
IEEE_SET_HALTING_MODE(FLAG,HALTING)	Controls continuation or halting on exceptions.
IEEE_SET_STATUS(STATUS_VALUE)	Restore the state of the floating point environment.

The module IEEE_ARITHMETIC contains the following nonelemental subroutines.

Subroutine	Description
IEEE_GET_ROUNDING_MODE(ROUND_VAL)	Get the current IEEE rounding mode.
IEEE_SET_ROUNDING_MODE(ROUND_VAL)	Set the current IEEE rounding mode.

2.2.2 C Binding Module

The Fortran 2000 draft standard provides a means of referencing C language procedures. The `ISO_C_BINDING` module defines three support procedures as intrinsic module functions. Accessing these functions requires

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_LOC, C_PTR, C_ASSOCIATED
```

in the calling routine. The procedures defined in the module are

Function	Description
`C_LOC(X)`	Returns the C address of the argument
`C_ASSOCIATED(C_PTR_1 [, C_PTR_2])`	Indicates the association status of `C_PTR_1` or indicates whether `C_PTR_1` and `C_PTR_2` are associated with the same entity.
`C_F_POINTER(CPTR, FPTR [, SHAPE])`	Associates a pointer with the target of a C pointer and specifies its shape.

For details on the `ISO_C_BINDING` intrinsic module, see Chapter 15 of the Fortran 2000 draft standard at http://www.j3-fortran.org/.

2.3 Non-Standard Fortran 95 Intrinsic Functions

The following functions are considered intrinsics by the `f95` compiler, but are not part of the Fortran 95 standard.

2.3.1 Basic Linear Algebra Functions (BLAS)

When compiling with `-xknown_lib=blas`, the compiler will recognize calls to the following routines as intrinsics and will optimize for and link to the Sun Performance Library implementation. The compiler will ignore user-supplied versions of these routines.

TABLE 2-2 BLAS Intrinsics

Function	Description
CAXPY	Product of a scalar and a vector plus a vector
DAXPY	
SAXPY	
ZAXPY	
CCOPY	Copy a vector
DCOPY	
SCOPY	
ZCOPY	
CDOTC	Dot product (inner product)
CDOTU	
DDOT	
SDOT	
ZDOTC	
ZDOTU	
CSCAL	Scale a vector
DSCAL	
SSCAL	
ZSCAL	

See the *Sun Performance Library User's Guide* for more information on these routines.

2.3.2 Interval Arithmetic Intrinsic Functions

The following table lists intrinsic functions that are recognized by the compiler when compiling for interval arithmetic (`-xia`). For details, see the *Fortran 95 Interval Arithmetic Programming Reference.*

DINTERVAL	DIVIX	INF	INTERVAL
ISEMPTY	MAG	MID	MIG
NDIGITS	QINTERVAL	SINTERVAL	SUP

VDABS	VDACOS	VDASIN	VDATAN
VDATAN2	VDCEILING	VDCOS	VDCOSH
VDEXP	VDFLOOR	VDINF	VDINT
VDISEMPTY	VDLOG	VDLOG10	VDMAG
VDMID	VDMIG	VDMOD	VDNINT
VDSIGN	VDSIN	VDSINH	VDSQRT
VDSUP	VDTAN	VDTANH	VDWID
VQABS	VQCEILING	VQFLOOR	VQINF
VQINT	VQISEMPTY	VQMAG	VQMID
VQMIG	VQNINT	VQSUP	VQWID
VSABS	VSACOS	VSASIN	VSATAN
VSATAN2	VSCEILING	VSCOS	VSCOSH
VSEXP	VSFLOOR	VSINF	VSINT
VSISEMPTY	VSLOG	VSLOG10	VSMAG
VSMID	VSMIG	VSMOD	VSNINT
VSSIGN	VSSIN	VSSINH	VSSQRT
VSSUP	VSTAN	VSTANH	VSWID
WID			

2.3.3 Other Vendor Intrinsic Functions

The `f95` compiler recognizes a variety of legacy intrinsic functions that were defined by Fortran compilers from other vendors, including Cray Research, Inc. These are obsolete and their use should be avoided.

TABLE 2-3 Intrinsic Functions From Cray CF90 and Other Compilers

Function	Arguments	Description
CLOC	([C=]c)	Obtains the address of a character object
COMPL	([I=]i)	Bit-by-bit complement of a word. Use NOT(i) instead
COT	([X=]x)	Generic cotangent. (Also: DCOT, QCOT)
CSMG	([I=]i,[J=]j,[K=]k)	Conditional Scalar Merge
DSHIFTL	([I=]i,[J=]j,[K=]k)	Double-object left shift of i and j by k bits
DSHIFTR	([I=]i,[J=]j,[K=]k)	Double-object right shift of i and j by k bits

TABLE 2-3 Intrinsic Functions From Cray CF90 and Other Compilers *(Continued)*

Function	Arguments	Description
EQV	([I=]i,[J=]j)	Logical equivalence. Use IOER(i,j) instead.
FCD	([I=]i,[J=]j)	Constructs a character pointer
GETPOS	([I=]i)	Obtains file position
IBCHNG	([I=]i, [POS=]j)	Generic function to change specified bit in a word.
ISHA	([I=]i, [SHIFT=]j)	Generic arithmetic shift
ISHC	([I=]i, [SHIFT=]j)	Generic circular shift
ISHL	([I=]i, [SHIFT=]j)	Generic left shift
LEADZ	([I=]i)	Counts number of leading 0 bits
LENGTH	([I=]i)	Returns the number of Cray words successfully transferred
LOC	([I=]i)	Returns the address of a variable (See Section 1.4.32, "loc: Return the Address of an Object" on page 1-51)
NEQV	([I=]i,[J=]j)	Logical non-equivalence. Use IOER(i,j) instead.
POPCNT	([I=]i)	Counts number of bits set to 1
POPPAR	([I=]i)	Computes bit popluation parity
SHIFT	([I=]i,[J=]j)	Shift left circular. Use ISHFT(i,j) or ISHFTC(i,j,k) instead.
SHIFTA	([I=]i,[J=]j)	Arithmetic shift with sign extension.
SHIFTL	([I=]i,[J=]j)	Shift left with zero fill. Use ISHFT(i,j) or ISHFTC(i,j,k) instead.
SHIFTR	([I=]i,[J=]j)	Shift right with zero fill. Use ISHFT(i,j) or ISHFTC(i,j,k) instead.
TIMEF	()	Returns elapsed time since the first call
UNIT	([I=]i)	Returns status of BUFFERIN or BUFFEROUT
XOR	([I=]i,[J=]j)	Logical exclusive OR. Use IOER(i,j) instead.

See also Chapter 3 for a list of VMS Fortran 77 intrinsics.

2.3.4 Other Extensions

The Fortran 95 compiler recognizes the following additional intrinsic functions:

2.3.4.1 MPI_SIZEOF

`MPI_SIZEOF( x, size, error)`

Returns the size in bytes of the machine representation of the given variable, *x*. If *x* is an array, it returns the size of the base element and not the size of the whole array

x	input; variable or array of arbitrary type
size	output; integer; size in bytes of *x*
error	output; integer; set to an error code if an error detected, zero otherwise

2.3.4.2 Memory Functions

Memory allocation, reallocation, and deallocation functions `malloc()`, `realloc()`, and `free()` are implemented as `f95` intrinsics. See Section 1.4.35, "`malloc`, `malloc64`, `realloc`, `free`: Allocate/Reallocate/Deallocate Memory" on page 1-55 for details.

CHAPTER 3

FORTRAN 77 and VMS Intrinsic Functions

This chapter lists the set of FORTRAN 77 intrinsic functions accepted by `f95` and is provided to aid migration of programs from legacy FORTRAN 77 to Fortran 95.

`f95` recognizes as intrinsic functions all the FORTRAN 77 and VMS functions listed in this chapter, along with all the Fortran 95 functions listed in the previous chapter. As an aid to migrating legacy FORTRAN 77 programs to `f95`, compiling with `-f77=intrinsics` causes the compiler to recognize only FORTRAN 77 and VMS functions as intrinsics, and not the Fortran 95 intrinsics.

Intrinsic functions that are Sun extensions of the ANSI FORTRAN 77 standard are marked with ✠. Programs using non-standard intrinsics and library functions may not be portable to other platforms.

Intrinsic functions have *generic* and *specific* names when they accept arguments of more than one data type. In general, the *generic* name returns a value with the same data type as its argument. However, there are exceptions such as the type conversion functions (TABLE 3-2) and the inquiry functions (TABLE 3-7). The function may also be called by one of its *specific* names to handle a specific argument data type.

With functions that work on more than one data item (e.g. `sign(a1,a2)`), all the data arguments must be the same type.

In the following tables, the FORTRAN 77 intrinsic functions are listed by:

- Intrinsic Function –description of what the function does
- Definition – a mathematical definition
- No. of Args. – number of arguments the function accepts
- Generic Name – the function's generic name
- Specific Names – the function's specific names
- Argument Type – data type associated with each specific name
- Function Type – data type returned for specific argument data type

Note – Compiler option `-xtypemap` changes the default sizes of variables and has an affect on intrinsic references. See Section 3.4, "Remarks" on page 3-13, and the discussion of default sizes and alignment in the *Fortran User's Guide*.

3.1 Arithmetic and Mathematical Functions

This section details arithmetic, type conversion, trigonometric, and other functions. "a" stands for a function's single argument, "a1" and "a2" for the first and second arguments of a two argument function, and "ar" and "ai" for the real and imaginary parts of a function's complex argument.

3.1.1 Arithmetic Functions

TABLE 3-1 Fortran 77 Arithmetic Functions

Intrinsic Function	Definition	No. of Args.	Generic Name	Specific Names	Argument Type	Function Type
Absolute value *See Note (6).*	|a| = $(ar^2+ai^2)^{1/2}$	1	ABS	IABS	INTEGER	INTEGER
				ABS	REAL	REAL
				DABS	DOUBLE	DOUBLE
				CABS	COMPLEX	REAL
				QABS ✠	REAL*16	REAL*16
				ZABS ✠	DOUBLE COMPLEX	DOUBLE
				CDABS ✠	DOUBLE COMPLEX	DOUBLE
				CQABS ✠	COMPLEX*32	REAL*16
Truncation *See Note (1).*	int(a)	1	AINT	AINT	REAL	REAL
				DINT	DOUBLE	DOUBLE
				QINT ✠	REAL*16	REAL*16
Nearest whole number	int(a+.5) if a ≥ 0 int(a-.5) if a < 0	1	ANINT	ANINT	REAL	REAL
				DNINT	DOUBLE	DOUBLE
				QNINT ✠	REAL*16	REAL*16
Nearest integer	int(a+.5) if a ≥ 0 int(a-.5) if a < 0	1	NINT	NINT	REAL	INTEGER
				IDNINT	DOUBLE	INTEGER
				IQNINT ✠	REAL*16	INTEGER

TABLE 3-1 Fortran 77 Arithmetic Functions *(Continued)*

Intrinsic Function	Definition	No. of Args.	Generic Name	Specific Names	Argument Type	Function Type
Remainder *See Note (1).*	a1-int(a1/a2)*a2	2	MOD	MOD	INTEGER	INTEGER
				AMOD	REAL	REAL
				DMOD	DOUBLE	DOUBLE
				QMOD ✠	REAL*16	REAL*16
Transfer of sign	|a1| if a2 ≥ 0 -|a1| if a2 < 0	2	SIGN	ISIGN	INTEGER	INTEGER
				SIGN	REAL	REAL
				DSIGN	DOUBLE	DOUBLE
				QSIGN ✠	REAL*16	REAL*16
Positive difference	a1-a2 if a1 > a2 0 if a1 ≤ a2	2	DIM	IDIM	INTEGER	INTEGER
				DIM	REAL	REAL
				DDIM	DOUBLE	DOUBLE
				QDIM ✠	REAL*16	REAL*16
Double and quad products	a1 * a2	2	–	DPROD	REAL	DOUBLE
				QPROD ✠	DOUBLE	REAL*16
Choosing largest value	max(a1, a2, …)	≥2	MAX	MAX0	INTEGER	INTEGER
				AMAX1	REAL	REAL
				DMAX1	DOUBLE	DOUBLE
				QMAX1 ✠	REAL*16	REAL*16
			AMAX0	AMAX0	INTEGER	REAL
			MAX1	MAX1	REAL	INTEGER
Choosing smallest value	min(a1, a2, …)	≥2	MIN	MIN0	INTEGER	INTEGER
				AMIN1	REAL	REAL
				DMIN1	DOUBLE	DOUBLE
				QMIN1 ✠	REAL*16	REAL*16
			AMIN0	AMIN0	INTEGER	REAL
			MIN1	MIN1	REAL	INTEGER

3.1.2 Type Conversion Functions

TABLE 3-2 Fortran 77 Type Conversion Functions

Conversion to	No. of Args	Generic Name	Specific Names	Argument Type	Function Type
INTEGER *See Note (1) .*	1	INT	-	INTEGER	INTEGER
			INT	REAL	INTEGER
			IFIX	REAL	INTEGER
			IDINT	DOUBLE	INTEGER
			-	COMPLEX	INTEGER
			-	COMPLEX*16	INTEGER
			-	COMPLEX*32	INTEGER
			IQINT ✠	REAL*16	INTEGER
REAL *See Note (2) .*	1	REAL	REAL	INTEGER	REAL
			FLOAT	INTEGER	REAL
			-	REAL	REAL
			SNGL	DOUBLE	REAL
			SNGLQ ✠	REAL*16	REAL
			-	COMPLEX	REAL
			-	COMPLEX*16	REAL
			-	COMPLEX*32	REAL
			FLOATK	INTEGER*8	REAL*4
DOUBLE *See Note (3).*	1	DBLE	DBLE	INTEGER	DOUBLE PRECISION
			DFLOAT	INTEGER	DOUBLE PRECISION
			DFLOATK	INTEGER*8	DOUBLE PRECISION
			DREAL ✠	REAL	DOUBLE PRECISION
			-	DOUBLE	DOUBLE PRECISION
			-	COMPLEX	DOUBLE PRECISION
			-	COMPLEX*16	DOUBLE PRECISION
			-	REAL*16	DOUBLE PRECISION
			-	COMPLEX*32	DOUBLE PRECISION
			DBLEQ ✠	REAL*16	DOUBLE PRECISION
			-	COMPLEX*32	DOUBLE PRECISION

TABLE 3-2 Fortran 77 Type Conversion Functions *(Continued)*

Conversion to	No. of Args	Generic Name	Specific Names	Argument Type	Function Type
REAL*16 *See Note (3').*	1	QREAL✠	QREAL ✠	INTEGER	REAL*16
		QEXT ✠	QFLOAT ✠	INTEGER	REAL*16
			–	REAL	REAL*16
			QEXT ✠	INTEGER	REAL*16
			QEXTD ✠	DOUBLE	REAL*16
			–	REAL*16	REAL*16
			–	COMPLEX	REAL*16
			–	COMPLEX*16	REAL*16
			–	COMPLEX*32	REAL*16
COMPLEX *See Notes (4) and (8).*	1 or 2	CMPLX	–	INTEGER	COMPLEX
			–	REAL	COMPLEX
			–	DOUBLE	COMPLEX
			–	REAL*16	COMPLEX
			–	COMPLEX	COMPLEX
			–	COMPLEX*16	COMPLEX
			–	COMPLEX*32	COMPLEX
DOUBLE COMPLEX *See Note (8).*	1 or 2	DCMPLX✠	–	INTEGER	DOUBLE COMPLEX
			–	REAL	DOUBLE COMPLEX
			–	DOUBLE	DOUBLE COMPLEX
			–	REAL*16	DOUBLE COMPLEX
			–	COMPLEX	DOUBLE COMPLEX
			–	COMPLEX*16	DOUBLE COMPLEX
			–	COMPLEX*32	DOUBLE COMPLEX
COMPLEX*32 *See Note (8).*	1 or 2	QCMPLX✠	–	INTEGER	COMPLEX*32
			–	REAL	COMPLEX*32
			–	DOUBLE	COMPLEX*32
			–	REAL*16	COMPLEX*32
			–	COMPLEX	COMPLEX*32
			–	COMPLEX*16	COMPLEX*32
			–	COMPLEX*32	COMPLEX*32
INTEGER *See Note (5).*	1	–	ICHAR	CHARACTER	INTEGER
		–	IACHAR ✠		
CHARACTER *See Note (5).*	1	–	CHAR	INTEGER	CHARACTER
		–	ACHAR ✠		

On an ASCII platform, including Sun systems:

- ACHAR is a nonstandard synonym for CHAR
- IACHAR is a nonstandard synonym for ICHAR

On a non-ASCII platform, ACHAR and IACHAR were intended to provide a way to deal directly with ASCII.

3.1.3 Trigonometric Functions

TABLE 3-3 Fortran 77 Trigonometric Functions

Intrinsic Function	Definition	Args	Generic Name	Specific Names	Argument Type	Function Type
Sine *See Note (7).*	sin(a)	1	SIN	SIN	REAL	REAL
				DSIN	DOUBLE	DOUBLE
				QSIN ✠	REAL*16	REAL*16
				CSIN	COMPLEX	COMPLEX
				ZSIN ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CDSIN ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CQSIN ✠	COMPLEX*32	COMPLEX*32
Sine (degrees) *See Note (7).*	sin(a)	1	SIND ✠	SIND ✠	REAL	REAL
				DSIND ✠	DOUBLE	DOUBLE
				QSIND ✠	REAL*16	REAL*16
Cosine *See Note (7).*	cos(a)	1	COS	COS	REAL	REAL
				DCOS	DOUBLE	DOUBLE
				QCOS ✠	REAL*16	REAL*16
				CCOS	COMPLEX	COMPLEX
				ZCOS ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CDCOS ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CQCOS ✠	COMPLEX*32	COMPLEX*32
Cosine (degrees) *See Note (7).*	cos(a)	1	COSD ✠	COSD ✠	REAL	REAL
				DCOSD ✠	DOUBLE	DOUBLE
				QCOSD ✠	REAL*16	REAL*16
Tangent *See Note (7).*	tan(a)	1	TAN	TAN	REAL	REAL
				DTAN	DOUBLE	DOUBLE
				QTAN ✠	REAL*16	REAL*16
Tangent (degrees) *See Note (7).*	tan(a)	1	TAND ✠	TAND ✠	REAL	REAL
				DTAND ✠	DOUBLE	DOUBLE
				QTAND ✠	REAL*16	REAL*16
Arcsine *See Note (7).*	arcsin(a)	1	ASIN	ASIN	REAL	REAL
				DASIN	DOUBLE	DOUBLE
				QASIN ✠	REAL*16	REAL*16

TABLE 3-3 Fortran 77 Trigonometric Functions *(Continued)*

Intrinsic Function	Definition	Args	Generic Name	Specific Names	Argument Type	Function Type
Arcsine (degrees) *See Note (7).*	arcsin(a)	1	ASIND ✠	ASIND ✠	REAL	REAL
				DASIND ✠	DOUBLE	DOUBLE
				QASIND ✠	REAL*16	REAL*16
Arccosine *See Note (7).*	arccos(a)	1	ACOS	ACOS	REAL	REAL
				DACOS	DOUBLE	DOUBLE
				QACOS ✠	REAL*16	REAL*16
Arccosine (degrees) *See Note (7).*	arccos(a)	1	ACOSD ✠	ACOSD ✠	REAL	REAL
				DACOSD ✠	DOUBLE	DOUBLE
				QACOSD ✠	REAL*16	REAL*16
Arctangent *See Note (7).*	arctan(a)	1	ATAN	ATAN	REAL	REAL
				DATAN	DOUBLE	DOUBLE
				QATAN ✠	REAL*16	REAL*16
	arctan (a1/a2)	2	ATAN2	ATAN2	REAL	REAL
				DATAN2	DOUBLE	DOUBLE
				QATAN2 ✠	REAL*16	REAL*16
Arctangent (degrees) *See Note (7).*	arctan(a)	1	ATAND ✠	ATAND ✠	REAL	REAL
				DATAND ✠	DOUBLE	DOUBLE
				QATAND ✠	REAL*16	REAL*16
	arctan (a1/a2)	2	ATAN2D✠	ATAN2D ✠	REAL	REAL
				DATAN2D ✠	DOUBLE	DOUBLE
				QATAN2D ✠	REAL*16	REAL*16
Hyperbolic Sine *See Note (7).*	sinh(a)	1	SINH	SINH	REAL	REAL
				DSINH	DOUBLE	DOUBLE
				QSINH ✠	REAL*16	REAL*16
Hyperbolic Cosine *See Note (7).*	cosh(a)	1	COSH	COSH	REAL	REAL
				DCOSH	DOUBLE	DOUBLE
				QCOSH ✠	REAL*16	REAL*16
Hyperbolic Tangent *See Note (7).*	tanh(a)	1	TANH	TANH	REAL	REAL
				DTANH	DOUBLE	DOUBLE
				QTANH ✠	REAL*16	REAL*16

3.1.4 Other Mathematical Functions

TABLE 3-4 Other Fortran 77 Mathematical Functions

Intrinsic Function	Definition	No. of Args.	Generic Name	Specific Names	Argument Type	Function Type
Imaginary part of a complex number *See Note (6).*	ai	1	IMAG	AIMAG	COMPLEX	REAL
				DIMAG ✠	DOUBLE COMPLEX	DOUBLE
				QIMAG ✠	COMPLEX*32	REAL*16
Conjugate of a complex number *See Note (6).*	(ar, -ai)	1	CONJG	CONJG	COMPLEX	COMPLEX
				DCONJG ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				QCONJG ✠	COMPLEX*32	COMPLEX*32
Square root	a**(1/2)	1	SQRT	SQRT	REAL	REAL
				DSQRT	DOUBLE	DOUBLE
				QSQRT ✠	REAL*16	REAL*16
				CSQRT	COMPLEX	COMPLEX
				ZSQRT ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CDSQRT ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CQSQRT ✠	COMPLEX*32	COMPLEX*32
Cube root *See Note(8′).*	a**(1/3)	1	CBRT	CBRT ✠	REAL	REAL
				DCBRT ✠	DOUBLE	DOUBLE
				QCBRT ✠	REAL*16	REAL*16
				CCBRT ✠	COMPLEX	COMPLEX
				ZCBRT ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CDCBRT ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CQCBRT ✠	COMPLEX*32	COMPLEX*32
Exponential	e**a	1	EXP	EXP	REAL	REAL
				DEXP	DOUBLE	DOUBLE
				QEXP ✠	REAL*16	REAL*16
				CEXP	COMPLEX	COMPLEX
				ZEXP ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CDEXP ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CQEXP ✠	COMPLEX*32	COMPLEX*32

TABLE 3-4 Other Fortran 77 Mathematical Functions *(Continued)*

Intrinsic Function	Definition	No. of Args.	Generic Name	Specific Names	Argument Type	Function Type
Natural logarithm	log(a)	1	LOG	ALOG	REAL	REAL
				DLOG	DOUBLE	DOUBLE
				QLOG ✠	REAL*16	REAL*16
				CLOG	COMPLEX	COMPLEX
				ZLOG ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CDLOG ✠	DOUBLE COMPLEX	DOUBLE COMPLEX
				CQLOG ✠	COMPLEX*32	COMPLEX*32
Common logarithm	log10(a)	1	LOG10	ALOG10	REAL	REAL
				DLOG10	DOUBLE	DOUBLE
				QLOG10 ✠	REAL*16	REAL*16
Error function *(See note below)*	erf(a)	1	ERF	ERF ✠	REAL	REAL
				DERF ✠	DOUBLE	DOUBLE
Error function	1.0 - erf(a)	1	ERFC	ERFC ✠	REAL	REAL
				DERFC ✠	DOUBLE	DOUBLE

- The error function: *2/sqrt(pi)* x integral from *0* to *a* of *exp(-t*t) dt*

3.2 Character Functions

TABLE 3-5 Fortran 77 Character Functions

Intrinsic Function	Definition	No. of Args.	Specific Names	Argument Type	Function Type
Conversion *See Note (5).*	Conversion to character	1	CHAR	INTEGER	CHARACTER
	Conversion to integer		ACHAR ✠		
	See also: TABLE 3-2.	1	ICHAR	CHARACTER	INTEGER
			IACHAR ✠		
Index of a substring	Location of substring a2 in string a1 *See Note (10).*	2	INDEX	CHARACTER	INTEGER
Length	Length of character entity *See Note (11).*	1	LEN	CHARACTER	INTEGER

TABLE 3-5 Fortran 77 Character Functions *(Continued)*

Intrinsic Function	Definition	No. of Args.	Specific Names	Argument Type	Function Type
Lexically greater than or equal	a1 ≥ a2 *See Note (12).*	2	LGE	CHARACTER	LOGICAL
Lexically greater than	a1 > a2 *See Note (12).*	2	LGT	CHARACTER	LOGICAL
Lexically less than or equal	a1 ≤ a2 *See Note (12).*	2	LLE	CHARACTER	LOGICAL
Lexically less than	a1 < a2 *See Note (12).*	2	LLT	CHARACTER	LOGICAL

On an ASCII platform (including Sun systems):

- ACHAR is a nonstandard synonym for CHAR
- IACHAR is a nonstandard synonym for ICHAR

On a non-ASCII platform, ACHAR and IACHAR were intended to provide a way to deal directly with ASCII.

3.3 Miscellaneous Functions

Other miscellaneous functions include bitwise functions, environmental inquiry functions, and memory allocation and deallocation functions.

3.3.1 Bit Manipulation ✠

None of these functions are part of the FORTRAN 77 Standard.

TABLE 3-6 Fortran 77 Bitwise Functions

Bitwise Operations	No. of Args.	Specific Name	Argument Type	Function Type
Complement	1	NOT	INTEGER	INTEGER
And	2	AND	INTEGER	INTEGER
	2	IAND	INTEGER	INTEGER
Inclusive or	2	OR	INTEGER	INTEGER
	2	IOR	INTEGER	INTEGER

TABLE 3-6 Fortran 77 Bitwise Functions *(Continued)*

Bitwise Operations	No. of Args.	Specific Name	Argument Type	Function Type
Exclusive or	2	XOR	INTEGER	INTEGER
	2	IEOR	INTEGER	INTEGER
Shift *See Note (14).*	2	ISHFT	INTEGER	INTEGER
Left shift *See Note (14).*	2	LSHIFT	INTEGER	INTEGER
Right shift *See Note (14).*	2	RSHIFT	INTEGER	INTEGER
Logical right shift *See Note (14).*	2	LRSHFT	INTEGER	INTEGER
Circular shift	3	ISHFTC	INTEGER	INTEGER
Bit extraction	3	IBITS	INTEGER	INTEGER
Bit set	2	IBSET	INTEGER	INTEGER
Bit test	2	BTEST	INTEGER	LOGICAL
Bit clear	2	IBCLR	INTEGER	INTEGER

The above functions are available as intrinsic or extrinsic functions. See also the discussion of the library bit manipulation routines in the *Fortran Library Reference* manual.

3.3.2 Environmental Inquiry Functions ✠

None of these functions are part of the FORTRAN 77 Standard.

TABLE 3-7 Fortran 77 Environmental Inquiry Functions

Definition	No. of Args.	Generic Name	Argument Type	Function Type
Base of Number System	1	EPBASE	INTEGER	INTEGER
			REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER
Number of Significant Bits	1	EPPREC	INTEGER	INTEGER
			REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER
Minimum Exponent	1	EPEMIN	REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER
Maximum Exponent	1	EPEMAX	REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER
Least Nonzero Number	1	EPTINY	REAL	REAL
			DOUBLE	DOUBLE
			REAL*16	REAL*16
Largest Number Representable	1	EPHUGE	INTEGER	INTEGER
			REAL	REAL
			DOUBLE	DOUBLE
			REAL*16	REAL*16
Epsilon *See Note (16).*	1	EPMRSP	REAL	REAL
			DOUBLE	DOUBLE
			REAL*16	REAL*16

3.3.3 Memory ✠

None of these functions are part of the FORTRAN 77 Standard.

TABLE 3-8 Fortran 77 Memory Functions

Intrinsic Function	Definition	No. of Args	Specific Name	Argument Type	Function Type
Location	Address of *See Note (17).*	1	LOC	*Any*	INTEGER*4 INTEGER*8
Allocate	Allocate memory and return address. *See Note (17).*	1	MALLOC MALLOC64	INTEGER*4 INTEGER*8	INTEGER INTEGER*8
Deallocate	Deallocate memory allocated by MALLOC. *See Note (17).*	1	FREE	*Any*	-
Size	Return the size of the argument in bytes. *See Note (18).*	1	SIZEOF	*Any expression*	INTEGER

3.4 Remarks

The following remarks apply to all of the intrinsic function tables in this chapter.

- The abbreviation DOUBLE stands for DOUBLE PRECISION.
- An intrinsic that takes INTEGER arguments accepts INTEGER*2, INTEGER*4, or INTEGER*8.
- INTEGER intrinsics that take INTEGER arguments return values of INTEGER type determined as follows. Note that the -xtypemap option may alter the default sizes of actual arguments:
  - mod sign dim max min and iand or ior xor ieor — size of the value returned is the largest of the sizes of the arguments.
  - abs ishft lshift rshift lrshft ibset btest ivclr ishftc ibits — size of the value returned is the size of the first argument.
  - int epbase epprec — size of the value returned is the size of default INTEGER.
  - ephuge — size of the value returned is the size of the default INTEGER, or the size of the argument, whichever is largest.

- Options that change the default data sizes also change the way some intrinsics are used. For example, with `-dbl` in effect, a call to `ZCOS` with a `DOUBLE COMPLEX` argument will automatically become a call to `CQCOS` because the argument has been promoted to `COMPLEX*32`. The following functions have this capability:

  ```
  aimag alog amod cabs ccbrt ccos cdabs cdcbrt cdcos cdexp
  cdlog cdsin cdsqrt cexp clog csin csqrt dabs dacos dacosd
  dasin dasind datan datand dcbrt dconjg dcos dcosd dcosh
  ddim derf derfc dexp dimag dint dlog dmod dnint dprod dsign
  dsin dsind dsinh dsqrt dtan dtand dtanh idnint iidnnt
  jidnnt zabs zcbrt zcos zexp zlog zsin zsqrt
  ```

- The following functions permit arguments of an integer or logical type of any size:

  ```
  and iand ieor iiand iieor iior inot ior jiand jieor jior
  jnot lrshft lshift not or rshift xor
  ```

- An intrinsic that is shown to return a *default* `REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `DOUBLE COMPLEX` value will return the prevailing type depending on certain compilation options. For example, if compiled with `-xtypemap=real:64,double:64`:
  - A call to a `REAL` function returns `REAL*8`
  - A call to a `DOUBLE PRECISION` function returns `REAL*8`
  - A call to a `COMPLEX` function returns `COMPLEX*16`
  - A call to a `DOUBLE COMPLEX` function returns `COMPLEX*16`

  Other options that alter the data sizes of default data types are `-r8` and `-dbl`, which also promote `DOUBLE` to `QUAD`. The `-xtypemap=` option provides more flexibility than these older compiler options and is preferred.
- A function with a generic name returns a value with the same type as the argument—except for type conversion functions, the nearest integer function, the absolute value of a complex argument, and others. If there is more than one argument, they must all be of the same type.
- If a function name is used as an actual argument, then it must be a specific name.
- If a function name is used as a dummy argument, then it does not identify an intrinsic function in the subprogram, and it has a data type according to the same rules as for variables and arrays.

3.4.1 Notes on Functions

Tables and notes 1 through 12 are based on the "Table of Intrinsic Functions," from *ANSI X3.9-1978 Programming Language FORTRAN*, with the Fortran extensions added.

(1) INT

If A is type integer, then INT(A) is A.

If A is type real or double precision, then:

if |A| < 1, then INT(A) is 0
if |A| ≥ 1, then INT(A) is the greatest integer that does not exceed the magnitude of A, and whose sign is the same as the sign of A. (Such a mathematical integer value may be too large to fit in the computer integer type.)

If A is type complex or double complex, then apply the above rule to the real part of A.

If A is type real, then IFIX(A) is the same as INT(A).

(2) REAL

If A is type real, then REAL(A) is A.

If A is type integer or double precision, then REAL(A) is as much precision of the significant part of A as a real datum can contain.

If A is type complex, then REAL(A) is the real part of A.

If A is type double complex, then REAL(A) is as much precision of the significant part of the real part of A as a real datum can contain.

(3) DBLE

If A is type double precision, then DBLE(A) is A.

If A is type integer or real, then DBLE(A) is as much precision of the significant part of A as a double precision datum can contain.

If A is type complex, then DBLE(A) is as much precision of the significant part of the real part of A as a double precision datum can contain.

If A is type COMPLEX*16, then DBLE(A) is the real part of A.

(3′) QREAL

If A is type REAL*16, then QREAL(A) is A.

If A is type integer, real, or double precision, then QREAL(A) is as much precision of the significant part of A as a REAL*16 datum can contain.

If A is type complex or double complex, then QREAL(A) is as much precision of the significant part of the real part of A as a REAL*16 datum can contain.

If A is type COMPLEX*16 or COMPLEX*32, then QREAL(A) is the real part of A.

(4) CMPLX

If A is type complex, then CMPLX(A) is A.

If A is type integer, real, or double precision, then CMPLX(A) is REAL(A) + 0i.

If A1 and A2 are type integer, real, or double precision, then CMPLX(A1,A2) is REAL(A1) + REAL(A2)*i.

If A is type double complex, then CMPLX(A) is REAL(DBLE(A)) + i*REAL(DIMAG(A)).

If CMPLX has two arguments, then they must be of the same type, and they may be one of integer, real, or double precision.

If CMPLX has one argument, then it may be one of integer, real, double precision, complex, COMPLEX*16, or COMPLEX*32.

(4') DCMPLX

If A is type COMPLEX*16, then DCMPLX(A) is A.

If A is type integer, real, or double precision, then DCMPLX(A) is DBLE(A) + 0i.

If A1 and A2 are type integer, real, or double precision, then DCMPLX(A1,A2) is DBLE(A1) + DBLE(A2)*i.

If DCMPLX has two arguments, then they must be of the same type, and they may be one of integer, real, or double precision.

If DCMPLX has one argument, then it may be one of integer, real, double precision, complex, COMPLEX*16, or COMPLEX*32.

(5) ICHAR

ICHAR(A) is the position of A in the collating sequence.

The first position is 0, the last is N-1, 0≤ICHAR(A)≤N-1, where N is the number of characters in the collating sequence, and A is of type character of length one.

CHAR and ICHAR are inverses in the following sense:

- ICHAR(CHAR(I)) = I, for 0≤I≤N-1
- CHAR(ICHAR(C)) = C, for any character C capable of representation in the processor

(6) COMPLEX

A COMPLEX value is expressed as an ordered pair of reals, (ar, ai), where ar is the real part, and ai is the imaginary part.

(7) Radians

All angles are expressed in radians, unless the "Intrinsic Function" column includes the *"(degrees)"* remark.

(8) COMPLEX Function

The result of a function of type COMPLEX is the principal value.

(8′) CBRT

If a is of COMPLEX type, CBRT results in COMPLEX RT1=(A, B), where: A ≥ 0.0, and -60 degrees ≤ arctan (B/A) < + 60 degrees.

Other two possible results can be evaluated as follows:

- RT2 = RT1 * (-0.5, square_root (0.75))
- RT3 = RT1 * (-0.5, square_root (0.75))

(9) Argument types

All arguments in an intrinsic function reference must be of the same type.

(10) INDEX

INDEX(X,Y) is the place in X where Y starts. That is, it is the starting position within character string X of the first occurrence of character string Y.

If Y does not occur in X, then INDEX(X,Y) is 0.

If LEN(X) < LEN(Y), then INDEX(X,Y) is 0.

INDEX returns default INTEGER*4 data. If compiling for a 64-bit environment, the compiler will issue a warning if the result overflows the INTEGER*4 data range. To use INDEX in a 64-bit environment with character strings larger than the INTEGER*4 limit (2 Gbytes), the INDEX function and the variables receiving the result must be declared INTEGER*8.

(11) LEN

LEN returns the declared length of the CHARACTER argument variable. The actual value of the argument is of no importance.

LEN returns default INTEGER*4 data. If compiling for a 64-bit environment, the compiler will issue a warning if the result overflows the INTEGER*4 data range. To use LEN in a 64-bit environment with character variables larger than the INTEGER*4 limit (2 Gbytes), the LEN function and the variables receiving the result must be declared INTEGER*8.

(12) Lexical Compare

LGE(X, Y) is true if X=Y, or if X follows Y in the collating sequence; otherwise, it is false.

LGT(X, Y) is true if X follows Y in the collating sequence; otherwise, it is false.

LLE(X, Y) is true if X=Y, or if X precedes Y in the collating sequence; otherwise, it is false.

LLT(X, Y) is true if X precedes Y in the collating sequence; otherwise, it is false.

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks.

(13) Bit Functions

There are other bitwise operations in VMS Fortran, but these are not implemented.

(14) Shift

LSHIFT shifts *a1* logically *left* by *a2* bits (inline code).

LRSHFT shifts *a1* logically *right* by *a2* bits (inline code).

RSHIFT shifts *a1* arithmetically *right* by *a2* bits.

ISHFT shifts *a1* logically *left* if *a2* > 0 and *right* if *a2* < 0.

The LSHIFT and RSHIFT functions are the Fortran analogs of the C << and >> operators. As in C, the semantics depend on the hardware.

The behavior of the shift functions with an out of range shift count is hardware dependent and generally unpredictable. In this release, shift counts larger than 31 result in hardware dependent behavior.

(15) Environmental inquiries

Only the type of the argument is significant.

(16) Epsilon

Epsilon is the least e, such that 1.0 + e ≠ 1.0.

(17) LOC, MALLOC, and FREE

The LOC function returns the address of a variable or of an external procedure. The function call MALLOC(*n*) allocates a block of at least *n* bytes, and returns the address of that block.

LOC returns default INTEGER*4 in 32-bit environments, INTEGER*8 in 64-bit environments.

MALLOC is a library function and not an intrinsic in FORTRAN 77. It too returns default INTEGER*4 in 32-bit environments, INTEGER*8 in 64-bit environments. However, MALLOC must be explicitly declared INTEGER*8 when compiling for 64-bit environments.

The value returned by LOC or MALLOC should be stored in variables typed POINTER, INTEGER*4, or INTEGER*8 in 64-bit environments. The argument to FREE must be the value returned by a previous call to MALLOC and hence should have data type POINTER, INTEGER*4, or INTEGER*8.

MALLOC64 always takes an INTEGER*8 argument (size of memory request in bytes) and always returns an INTEGER*8 value. Use this routine rather than MALLOC when compiling programs that must run in both 32-bit and 64-bit environments. The receiving variable must be declared either POINTER or INTEGER*8.

(18) SIZEOF

The SIZEOF intrinsic cannot be applied to arrays of an assumed size, characters of a length that is passed, or subroutine calls or names. SIZEOF returns default INTEGER*4 data. If compiling for a 64-bit environment, the compiler will issue a warning if the result overflows the INTEGER*4 data range. To use SIZEOF in a 64-bit environment with arrays larger than the INTEGER*4 limit (2 Gbytes), the SIZEOF function and the variables receiving the result must be declared INTEGER*8.

3.5 VMS Intrinsic Functions

This section lists VMS FORTRAN intrinsic routines recognized by f95. They are, of course, nonstandard. ✠

3.5.1 VMS Double-Precision Complex

TABLE 3-9 VMS Double-Precision Complex Functions

Generic Name	Specific Names	Function	Argument Type	Result Type
	CDABS	Absolute value	COMPLEX*16	REAL*8
	CDEXP	Exponential, e**a	COMPLEX*16	COMPLEX*16
	CDLOG	Natural log	COMPLEX*16	COMPLEX*16
	CDSQRT	Square root	COMPLEX*16	COMPLEX*16
	CDSIN	Sine	COMPLEX*16	COMPLEX*16
	CDCOS	Cosine	COMPLEX*16	COMPLEX*16
DCMPLX		Convert to DOUBLE COMPLEX	Any numeric	COMPLEX*16
	DCONJG	Complex conjugate	COMPLEX*16	COMPLEX*16
	DIMAG	Imaginary part of complex	COMPLEX*16	REAL*8
	DREAL	Real part of complex	COMPLEX*16	REAL*8

3.5.2 VMS Degree-Based Trigonometric

TABLE 3-10 VMS Degree-Based Trigonometric Functions

Generic Name	Specific Names	Function	Argument Type	Result Type
SIND		Sine	-	-
	SIND		REAL*4	REAL*4
	DSIND		REAL*8	REAL*8
	QSIND		REAL*16	REAL*16
COSD		Cosine	-	-
	COSD		REAL*4	REAL*4
	DCOSD		REAL*8	REAL*8
	QCOSD		REAL*16	REAL*16
TAND		Tangent	-	-
	TAND		REAL*4	REAL*4
	DTAND		REAL*8	REAL*8
	QTAND		REAL*16	REAL*16
ASIND		Arc sine	-	-
	ASIND		REAL*4	REAL*4
	DASIND		REAL*8	REAL*8
	QASIND		REAL*16	REAL*16
ACOSD		Arc cosine	-	-
	ACOSD		REAL*4	REAL*4
	DACOSD		REAL*8	REAL*8
	QACOSD		REAL*16	REAL*16
ATAND		Arc tangent	-	-
	ATAND		REAL*4	REAL*4
	DATAND		REAL*8	REAL*8
	QATAND		REAL*16	REAL*16
ATAN2D		Arc tangent of a1/a2	-	-
	ATAN2D		REAL*4	REAL*4
	DATAN2D		REAL*8	REAL*8
	QATAN2D		REAL*16	REAL*16

3.5.3 VMS Bit-Manipulation

TABLE 3-11 VMS Bit-Manipulation Functions

Generic Name	Specific Names	Function	Argument Type	Result Type
IBITS		From a1, initial bit a2, extract a3 bits	–	–
	IIBITS		INTEGER*2	INTEGER*2
	JIBITS		INTEGER*4	INTEGER*4
	KIBITS		INTEGER*8	INTEGER*8
ISHFT		Shift a1 logically by a2 bits; if a2 positive shift left, if a2 negative shift right	–	–
	IISHFT		INTEGER*2	INTEGER*2
	JISHFT		INTEGER*4	INTEGER*4
	KISHFT		INTEGER*8	INTEGER*8
ISHFTC		In a1, circular shift by a2 places, of right a3 bits	–	–
	IISHFTC		INTEGER*2	INTEGER*2
	JISHFTC		INTEGER*4	INTEGER*4
IAND		Bitwise AND of a1, a2	–	–
	IIAND		INTEGER*2	INTEGER*2
	JIAND		INTEGER*4	INTEGER*4
IOR		Bitwise OR of a1, a2	–	–
	IIOR		INTEGER*2	INTEGER*2
	JIOR		INTEGER*4	INTEGER*4
	KIOR		INTEGER*8	INTEGER*8
IEOR		Bitwise exclusive OR of a1, a2	–	–
	IIEOR		INTEGER*2	INTEGER*2
	JIEOR		INTEGER*4	INTEGER*4
	KIEOR		INTEGER*8	INTEGER*8
NOT		Bitwise complement	–	–
	INOT		INTEGER*2	INTEGER*2
	JNOT		INTEGER*4	INTEGER*4
	KNOT		INTEGER*8	INTEGER*8

TABLE 3-11 VMS Bit-Manipulation Functions *(Continued)*

Generic Name	Specific Names	Function	Argument Type	Result Type
IBSET		In a1, set bit a2 to 1; return new a1	–	–
	IIBSET		INTEGER*2	INTEGER*2
	JIBSET		INTEGER*4	INTEGER*4
	KIBSET		INTEGER*8	INTEGER*8
BTEST		If bit a2 of a1 is 1, return .TRUE.	–	–
	BITEST		INTEGER*2	INTEGER*2
	BJTEST		INTEGER*4	INTEGER*4
	BKTEST		INTEGER*8	INTEGER*8
IBCLR		In a1, set bit a2 to 0; return new a1	–	–
	IIBCLR		INTEGER*2	INTEGER*2
	JIBCLR		INTEGER*4	INTEGER*4
	KIBCLR		INTEGER*8	INTEGER*8

3.5.4 VMS Multiple Integer Types

The possibility of multiple integer types is not addressed by the Fortran Standard. The compiler copes with their existence by treating a specific INTEGER-to-INTEGER function name (IABS, and so forth) as a special sort of generic. The argument type is used to select the appropriate runtime routine name, which is not accessible to the programmer.

VMS Fortran takes a similar approach, but makes the specific names available.

TABLE 3-12 VMS Integer Functions

Specific Names	Function	Argument Type	Result Type
IIABS	Absolute value	INTEGER*2	INTEGER*2
JIABS		INTEGER*4	INTEGER*4
KIABS		INTEGER*8	INTEGER*8
IMAX0	Maximum [1]	INTEGER*2	INTEGER*2
JMAX0		INTEGER*4	INTEGER*4
IMIN0	Minimum [1]	INTEGER*2	INTEGER*2
JMIN0		INTEGER*4	INTEGER*4

TABLE 3-12 VMS Integer Functions *(Continued)*

Specific Names	Function	Argument Type	Result Type
IIDIM	Positive difference [2]	INTEGER*2	INTEGER*2
JIDIM		INTEGER*4	INTEGER*4
KIDIM		INTEGER*8	INTEGER*8
IMOD	Remainder of a1/a2	INTEGER*2	INTEGER*2
JMOD		INTEGER*4	INTEGER*4
IISIGN	Transfer of sign, |a1|* sign(a2)	INTEGER*2	INTEGER*2
JISIGN		INTEGER*4	INTEGER*4
KISIGN		INTEGER*8	INTEGER*8

1 There must be at least two arguments.

2 The positive difference is: a1-min(a1,a2))

Index

D

M

N

O

P

Q

R

S

T

U

V

W

X