i960® Processor Library Supplement

Order Number: 651231-004

Revision	Revision History	Date
-001	Original Issue.	02/96
-002	Revised for Release 5.1.	01/97
-003	Revised for Release 6.0.	12/97
-004	Revised for Release 6.5.	12/98

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Intel Corporation
PO Box 5937
Denver, CO 80217-9808

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office.

Contact your local sales office to obtain the latest specifications before placing your order.

printed on
recycled paper

Contents

Overview

1

This chapter introduces the libraries and this manual. It also identifies sources of detailed or supplemental information.

The i960® processor libraries ease application development by providing:

- interfaces to standard and custom execution environments
- C, C++, and assembly-language functions
- macro definitions and type declarations
- a variety of linkable files and library sources
- floating-point emulation libraries

Compatibility With Standards

The libraries provide standard and i960 processor-specific library and header files. The standard parts of the C libraries are compatible with the ANSI X3.159-1989 standard for the C language. Note, however, that the following ANSI C functions are implemented as stubs and do not return meaningful values.

```
clock                setlocale
localeconv           strcoll
mblen                strxfrm
mbstowcs             system
mbtowc               wcstombs
rename               wctomb
```

The C++ portion of the libraries include the Free Software Foundation's implementation of the C++ Iostream classes.

The i960 processor-specific parts of the libraries:

- provide for more efficient use of the Cx, Hx, Jx, Kx, Rx, Sx , and VH processor implementations

- emulate the KB processor's floating-point extensions
- include low-level libraries for the MON960-supported evaluation boards.

To make porting programs from other systems easier, the libraries also include selected functions defined in the IEEE Standard 1003.1-1988 Portable Operating System Interface for Computer Environments (POSIX), UNIX System Laboratories, Inc. System V Interface Definition (SVID), and other sources added for completeness. However, library functions do not necessarily fully conform to the POSIX standard.

For details on the POSIX standard, see the IEEE Standard 1003.1-1988, *IEEE Standard Portable Operating System Interface for Computer Environments*, by IEEE, Inc. For information on SVID, see the *System V Interface Definition*, by UNIX System Laboratories, Inc. The next section of this chapter provides ordering information for POSIX and SVID publications.

Deciding Which Libraries to Use

To select the appropriate libraries, startup code, and object files for your target environment and the particular i960 processor you are using, read Chapter 2.

Using Functions

If you are using functions and macros specific to the i960 architecture read Chapter 3 to learn about the non-ANSI header files and Chapter 4 to learn about non-ANSI run-time library functions. The standard ANSI C run-time library functions are described in *C: A Reference Manual*.

Retargeting the Libraries

To retarget the libraries for execution in your own hardware environment, to write additional functions needed for reentrant programs, and to find reference information on target system calls and other low-level, non-portable functions, read Chapters 1 and 5.

About This Manual

This *i960 Processor Library Supplement* is a supplement to Part 2 of *C: A Reference Manual*. The *i960 Processor Library Supplement* describes the processor-specific and board-specific libraries and header files. This manual does not describe the ANSI standard C libraries and header files which are described in *C: A Reference Manual*. For information on standard C libraries, see *C: A Reference Manual*, by Samuel P. Harbison and Guy L. Steele, Jr., published by Prentice Hall, 1991. This book is available from Intel under order number 480628.

Portions of this manual use materials reprinted and adapted from IEEE Standard 1003.1-1988, *IEEE Standard Portable Operating System Interface for Computer Environments*, copyright 1988 by The Institute of Electrical and Electronics Engineers with the permission of the IEEE Standards Department. Text appearing in this document adapted from IEEE Standard 1003.1-1988 does not represent the approved IEEE Standard. In the event of a discrepancy between the version in this manual and the original standard version, the original version takes precedence.

Throughout this manual, “ANSI” refers to ANSI X3.159-1989 standard for the C language.

Related Publications

For information on related publications, see *Getting Started with the i960 Processor Software Tools*.

Customer Service

For customer service information, see *Getting Started with the i960 Processor Software Tools.*

Copyrights

Refer to the *i960 Software Tools License Guide* for licensing and copyright statement.

Using the Libraries 2

This chapter tells you how to use the libraries provided with CTOOLS in your programs. If your program uses any library functions, you must:

- Include the header files to use the library function declarations and type and macro definitions. See the *i960 Processor Compiler User's Guide* for information on including the headers.
- Compile your source text to produce an object module compatible with the libraries.
- Link your application object modules to the appropriate libraries, as discussed in the following section. The *i960 Processor Software Utilities User's Guide* explains how to use the linker.

Linking Libraries and Object Modules

The libraries consist of a set of portable or high-level libraries and a set of primitive or low-level libraries for each of the i960 KA/SA, KB/SB, Cx, Jx, Hx, Rx, and VH processor variations. You can use functions from the high-level libraries without modification in many different execution environments.

However, many functions in the high-level libraries call functions in the low-level libraries. The low-level libraries are specific to the evaluation boards which support the Intel MON960 debug monitor. The profiling and the ghist960 support libraries are also provided for the IxWorks* runtime environment.

For execution in any other environment, you often have to rewrite or supplement the functions in the low-level libraries for your particular target environment.

The following sections discuss the different library files you can link with your application program.

Library Files

For complete information about library names, see the Library List section, below. The library files are named following this general scheme:

`lib[`*`abbr`*`][`*`arch`*`][`*`qualifier`*`].a`

- *abbr* is an abbreviation of a library name. For example:
  - — `c` contains the standard ANSI C functions.
  - — `m` contains the standard ANSI math functions.
  - — `h` contains the accelerated floating-point functions for processors without on-chip floating-point support.
  - — `ll` contains a MON960 low-level library.
  - — `i` contains the C++ Iostream library.
  - — `u` contains 64-bit integer support
- *arch* if present, indicates the processor(s) the library can be used for:
  - — `ca` for Cx, Hx, Jx, and VH processors.
  - — `jx` for Jx- and VH-tuned floating-point libraries.
  - — `ka` for KA and SA processors.
  - — `kb` for KB and SB processors.
  - — `rp` for Rx processors.
  - — If *arch* is not present, the library can be used for all architectures (e.g., `libll.a`).
- *qualifier*
  if present, means that the library was generated with specific compiler options. All libraries contain position-independent code (PIC). Additionally:
  - — `_p` or `p` means that the library contains position-independent data (PID).
  - — `_b` or `b` indicates a big-endian library for Cx, Hx, and Jx applications.
  - — `_e` or `e` indicates a PID and big-endian library for Cx, Hx, and Jx applications.

Note that the `libh` library was designed in such a way that it can be used with both PID and non-PID programs, even though it has no `p` qualifier in its name.

If your application is a PIC program (linked with the `-pc` or `-pb` linker option), all of your modules must be compiled with the compiler's PIC option (`-mpic` for gcc960; `-Gpc` for ic960). Otherwise, the linker generates a warning.

If your application is not a PIC program, you can link PIC and non-PIC modules.

If your application is a PID program (linked with the `-pd` or `-pb` linker option), all modules and libraries must be PID. In other words, your modules must be compiled with gcc960's `-mpid` or `-mpid-safe` options or ic960's `-Gpd` or `-Gpr` option and linked with the appropriate `_p` libraries. Otherwise, the linker generates a warning.

If your application is not a PID program, link only non-PID modules.

The low-level library for MON960-based targets is `libll.a`. This library contains the low-level libraries for evaluation boards that support the Intel MON960 debug monitor.

Use `libllp.a` for PIC/PID programs.

Use `libll.a` for non-PID programs.

Use `libllb.a` for big-endian programs.

Use `liblle.a` for PID, big-endian programs.

Note that the libraries are supplied using the ELF object module format. The linker will automatically convert the libraries to your selected object module format.

Intel provides versions of the low-level libraries specific to the i960 Rx processor, `libllrp.a` and `libllrpp.a`. Note that the i960 Rx processor does not support big endian byte order. Because of this, no big endian libraries are provided for the i960 Rx processor.

Library List

Table 2-1 explains the abbreviations found in the library listings. All libraries shipped with the compiler are listed below Table 2-1.

Table 2-1 Library Use Abbreviation Table

Abbreviation	Meaning
BE	Big-endian.
CA	Use for 80960Cx, Hx, Jx and VH applications.
FILE-SYSTEM	For profiling libraries. This library is for applications which have file system services such as read, write, open, and close calls available to them.
JX	Jx- and VH-tuned floating-point library.
KA	Use for KA and SA applications.
KB	Use for KB and SB applications.
NO-FILE-SYS	For profiling libraries. This library is for applications which do not have file system services such as read, write, open, and close calls available to them. If these calls are not supported, use libq. If the calls are supported, use libqf. If you are using the Intel MON960 debug monitor, use libqf which has file system support in it.
PID	The library contains position-independent data (PID).
RP	Use for 80960Rx applications.

The files in the left column below are in `I960BASE/lib` (ic960 interface), or in `G960BASE/lib` (gcc960 interface).

The usage of each library is abbreviated in the right-hand column.

crt Startup Files

Your linked program must contain startup code to initialize the execution environment and the libraries in the first module that executes. The libraries include the following startup modules:

```
crt960.o
crt960_p.o                PID
crt960_b.o                BE
crt960_e.o                PID,BE

crtrp.o                   RP
crtrp_p.o                 RP,PID
```

libi C++ IOstream Library

CTOOLS now provides the libraries listed below, which provide Free Software Foundation's implementation of C++ Iostream classes.

```
libica.a                  CA
libica_b.a                CA,BE
libica_e.a                CA,PID,BE
libica_p.a                CA,PID

libika.a                  KA
libika_p.a                KA,PID

libikb.a                  KB
libikb_p.a                KB,PID

libirp.a                  RP
libirp_p.a                RP,PID
```

The associated C++ header files are included in a separate sub-directory named `cxxinc` in the CTOOLS distribution.

libc ANSI Standard Library

This is the ANSI C standard library, in ELF format.

```
libcca.a                  CA
libcca_b.a                CA,BE
```

libcca_e.a	CA,PID,BE
libcca_p.a	CA,PID
libcka.a	KA
libcka_p.a	KA,PID
libckb.a	KB
libckb_p.a	KB,PID
libcrp.a	RP
libcrp_p.a	RP,PID

libm ANSI Math Functions

This library contains the ANSI C standard math functions.

The `libst.a` library provides minimal function definitions to resolve external references during linking without adding the unnecessary code for full floating-point functionality. Use this library instead of `libm`*`xx`*`.a` if your program does not perform any floating-point number operations. The functions in `libst.a` do nothing more than resolve external references, so you can link this library with PID programs, and with any architecture.

libmca.a	CA
libmca_b.a	CA,BE
libmca_e.a	CA,BE,PID
libmca_p.a	CA,PID
libmka.a	KA
libmka_p.a	KA,PID
libmkb.a	KB
libmkb_p.a	KB,PID
libmrp.a	RP
libmrp_p.a	RP,PID
libst.a	
libstb.a	BE
libste.a	BE,PID
libstp.a	PID

libstrp.a	RP
libstrpp.a	RP,PID

libh Floating-point Library

This is the floating-point arithmetic library. Note that all of the `libh` libraries can be used in either PIC/PID or non-PIC/PID applications.

This library contains accelerated floating-point functions for processors without on-chip floating-point support. These functions implement floating-point operations without using any floating-point instructions.

libhca.a	CA
libhca_b.a	CA,BE
libhca_e.a	CA,BE,PID
libhca_p.a	CA,PID
libhjx.a	JX
libhjx_b.a	JX,BE
libhjx_e.a	JX,BE,PID
libhjx_p.a	JX,PID
libhka.a	KA
libhka_p.a	KA,PID
libhrp.a	RP
libhrp_p.a	RP,PID

For information on these libraries, see Chapter 6.

libfp Alternate Floating-point Library

This is an alternate floating-point arithmetic library. This library cannot be used in PIC/PID applications. It can be used as a partial replacement for `libh`. It is somewhat faster than `libh` although less accurate.

libfp.a	KA/CA
libfpb.a	BE
libfpe.a	BE,PID
libfpp.a	PID
libfprp.a	RP
libfprpe.a	RP,PID

libq/libqf Profiling Libraries

These are the libraries supplied to support profile-driven optimization. See the discussion of profiling in your compiler manual for details.

```
libq.a                    NO-FILE-SYSTEM
libqb.a                   NO-FILE-SYSTEM,BE
libqe.a                   NO-FILE-SYSTEM,PID,BE
libqp.a                   PID,NO-FILE-SYSTEM

libqf.a                   FILE-SYSTEM
libqfb.a                  FILE-SYSTEM,BE
libqfe.a                  FILE-SYSTEM,PID,BE
libqfp.a                  PID,FILE-SYSTEM

libqrp.a                  RP,NO-FILE-SYSTEM
libqrpp.a                 RP,NO-FILE-SYSTEM,PID

libqfrp.a                 RP,FILE-SYSTEM
libqfrpp.a                RP,FILE-SYSTEM,PID

libixqrp.a                RP, NO-FILE-SYSTEM, IxWorks*
```

libll MON960 Low-level Support Library

This is the low-level support library for evaluation boards that support the Intel MON960 debug monitor.

```
libll.a
libllb.a                  BE
liblle.a                  PID,BE
libllp.a                  PID

libllrp.a                 RP
libllrpp.a                RP,PID
```

libmon Monitor Support Library

This provides a calls interface for benchmark timing, flash memory, and ghist960 programming.

```
libmn.a
libmnb.a                  BE
libmne.a                  PID,BE
libmnp.a                  PID
```

libmnrp.a	RP
libmnrpp.a	RP,PID

libhs ghist960 Support Library

This is the ghist960 support library.

libhs.a	
libhsb.a	BE
libhse.a	PID,BE
libhsp.a	PID
libhsrp.a	RP
libhsrpp.a	RP,PID
libixhsrp.a	RP, IxWorks

librom Flash Support Library

This is the flash support library. All libraries support serially re-usable programs.

librm.a	
librmb.a	BE
librme.a	PID,BE
librmp.a	PID
librmrp.a	RP
librmrpp.a	RP,PID

64-bit Integer Support Library

CTOOLS now includes support for 64-bit integers using the *long long* type. Changes have been made in the compiler to support the *long long* type and valid operations on the same. The CTOOLS distribution now includes a new library which implements the routines that are needed to support operations on 64-bit integers.

The standard C library- *libc* includes the following routines to support the *long long* type:

```
atoll  lltoa  lltoh  lltos  strtoll  strtoull  ulltoa
```

For a description of these routines please take a look at the corresponding routines for the type *long* in chapter 4 of this manual.

CTOOLS now provides the libraries listed below, which provide Free Software Foundation's implementation of several library routines required for *long long* support.

```
libuca.a                CA
libuca_b.a              CA,BE
libuca_e.a              CA,PID,BE
libuca_p.a              CA,PID

libuka.a                KA
libuka_p.a              KA,PID

libukb.a                KB
libukb_p.a              KB,PID

liburp.a                RP
liburp_p.a              RP,PID
```

C Linker Directive Files

See the *i960 Processor Software Utilities Guide* for more information on the linker (lnk960, gld960) and linker directive files.

```
cycx.ld            Cyclone Cx
cycxb.ld           Cyclone Cx,BE
cycxbfls.ld        Cyclone Cx,BE,flash
cycxfls.ld         Cyclone Cx,flash
cycxp.ld           Cyclone Cx,PID
cycxpfls.ld        Cyclone Cx,PID,flash

cyhx.ld            Cyclone Hx
cyhxfls.ld         Cyclone Hx,flash

cyjx.ld            Cyclone Jx
cyjxb.ld           Cyclone Jx,BE
cyjxbfls.ld        Cyclone Jx,BE,flash
cyjxfls.ld         Cyclone Jx,flash
cyjxp.ld           Cyclone Jx,PID
cyjxpfls.ld        Cyclone Jx,PID,flash

cykx.ld            Cyclone Kx
cykxp.ld           Cyclone Kx,PID
```

`cysx.ld`	`Cyclone Sx`
`cysxp.ld`	`Cyclone Sx,PID`
`cyrx.ld`	`Cyclone RP`
`cyrxp.ld`	`Cyclone RP,PID`
`cyrxfls.ld`	`Cyclone RP,flash`
`cyrxpfls.ld`	`Cyclone RP,flash,PID`
`cytx.ld`	`Cyclone RN/RM`
`cytxp.ld`	`Cyclone RN/RM,PID`
`cytxfls.ld`	`Cyclone RN/RM,flash`
`cytxpfls.ld`	`Cyclone RN/RM,flash,PID`
`cyvx.ld`	`Cyclone VH`
`cyvxp.ld`	`Cyclone VH,PID`
`cyvxfls.ld`	`Cyclone VH,flash`
`cyvxpfls.ld`	`Cyclone VH,flash,PID`

C++ Linker Directive Files

The compiler distribution includes the following new linker directive files. These linker directive files are meant to be used when linking in C++ modules using the ic960 driver to form an absolute file.

```
cycc.ld             Cyclone Cx
cyccb.ld            Cyclone Cx,BE
cyccbfls.ld         Cyclone Cx,BE,flash
cyccfls.ld          Cyclone Cx,flash
cyccp.ld            Cyclone Cx,PID
cyccpfls.ld         Cyclone Cx,PID,flash

cyhc.ld             Cyclone Hx
cyhcfls.ld          Cyclone Hx,flash

cyjc.ld             Cyclone Jx
cyjcb.ld            Cyclone Jx,BE
cyjcbfls.ld         Cyclone Jx,BE,flash
cyjcfls.ld          Cyclone Jx,flash
cyjcp.ld            Cyclone Jx,PID
cyjcpfls.ld         Cyclone Jx,PID,flash

cykc.ld             Cyclone Kx
cykcp.ld            Cyclone Kx,PID

cysc.ld             Cyclone Sx
cyscp.ld            Cyclone Sx,PID

cyrc.ld             Cyclone RP
cyrcp.ld            Cyclone RP,PID
cyrcfls.ld          Cyclone RP,flash
cyrcpfls.ld         Cyclone RP,flash,PID

cytc.ld             Cyclone RN/RM
cytcp.ld            Cyclone RN/RM,PID
cytcfls.ld          Cyclone RN/RM,flash
cytcpfls.ld         Cyclone RN/RM,flash,PID

cyvc.ld             Cyclone VH
cyvcp.ld            Cyclone VH,PID
cyvcfls.ld          Cyclone VH,flash
cyvcpfls.ld         Cyclone VH,flash,PID
```

These new linker directive files allocate the sections “ctors” and “dtors” to proper locations and request the linker to include the C++ standard libraries in the search path for unresolved externals. The standard C++ libraries are searched ahead of the standard C libraries. The “ctors” and “dtors” sections are used to initialize/destroy static objects.

When generating an absolute module targeted for a Cyclone Cx board with an i960 CA processor, you would use a command such as:

```
ic960 -Tcycx -ACA t1.c t2.c
```

To include C++ modules in the absolute file, use a command such as:

```
ic960 -Tcycc -ACA t1.cc t2.c
```

The argument `-Tcycc` instructs the compiler to generate code for a Cyclone Cx board and to link in the C++ Iostream class library. Note that the gcc960 invocation options are not affected and remain the same. Therefore, you can continue using a command such as:

```
gcc960 -Fcoff -Tmcycx -ACA t1.cc t2.c
```

gcc960 Configuration Files

```
mcycx.gld           Cyclone Cx
mcycxfls.gld        Cyclone Cx,flash
mcyhx.gld           Cyclone Hx
mcyhxfls.gld        Cyclone Hx,flash
mcyjx.gld           Cyclone Jx
mcyjxfls.gld        Cyclone Jx,flash
mcykx.gld           Cyclone Kx
mcyrx.gld           Cyclone RP
mcyrxfls.gld        Cyclone RP,flash
mcysx.gld           Cyclone Sx

mcytx.gld           Cyclone RM/RN
mcytxfls.gld        Cyclone RM/RN,flash

mcyvx.gld           Cyclone VH
mcyvxfls.gld        Cyclone VH,flash
```

Linking Sequence

The linking order of libraries and object modules in your program depends on the file sequence you specify on the linker command line or in the linker configuration file. See the linker chapter of the *i960 Processor Utilities User's Guide* for information on the linking sequence.

To correctly link and execute your program, you must use the following order when you specify startup modules, libraries, and your program modules for linking:

1. startup code
2. program modules
3. user-defined libraries, if any
4. profiling library, statistical profiler library, flash support library
5. C++ Iostream (if specified)
6. standard C library
7. 64-Bit integer support library
8. standard math library
9. low-level, board-specific library
10. accelerated floating-point library, for the i960 KA, SA, Cx, Hx, and Jx processors only.

Using the Floating-point Libraries

The i960 KB and SB microprocessors implement in hardware the full i960 floating-point instruction set. The i960 processor computational model is fully compatible with IEEE standard P754 and allows the compiler to generate efficient floating-point instruction sequences, reducing the amount of object code generated. Programs ported from environments that do not conform to the IEEE standard can behave unpredictably, especially when floating-point exceptions occur.

Note that to use `libfp.a`, you must link both `libfp.a` and `libh`*`xx`*`.a` into your application. Furthermore, `libfp.a` must be specified to the linker before `libh`*`xx`*`.a` is specified.

The `libm`*`xx`*`.a` and `libm`*`xx`*`_p.a` standard math libraries can use either floating-point instructions or simulated floating-point operations. Functions in `libmkb.a` and `libmkb_p.a`, for processors with on-chip floating-point support, use floating-point instructions implemented in the processor instruction set. Functions in `libmka.a`, `libmca.a`, `libmka_p.a` and `libmca_p.a`, for processors without on-chip floating-point support, call low-level functions in `libhka.a` and `libhca.a`. The `libh`*`xx`*`.a` functions simulate floating-point instructions and can be used with both PIC/PID and non-PIC/PID programs.

Floating-point functions in `libh`*`xx`* support all levels of precision supported by the i960 architecture, as follows:

- Single-precision functions use the float data type.
- Double-precision functions use the double data type. Hyperbolic functions are available in double precision only.
- Extended-precision functions use the long double data type.

Since the floating-point functions round computations to the nearest representable least-significant digit, results using different rounding modes can differ. You can use macros and functions from the `fpsl.h` header file to set the rounding mode.

The floating-point functions comply with the IEEE P754 standard specification on operations with Not-a-Number elements (NaNs). If the arguments to a function are invalid for the operation or involve a Signaling NaN (SNaN), a Quiet NaN (QNaN) is returned and the `FPX_INVOP` exception is flagged. Functions process and return QNaNs without flagging any exceptions.

See Chapter 6 for more information on the floating-point emulation libraries.

Since the i960 Cx/Hx/Jx processors do not implement the floating-point bits in the arithmetic controls (AC) register, your Cx/Hx/Jx program must reserve a word in memory to contain the AC floating-point bits. This memory location must be named `fpem_CA_AC`. For fastest memory access, locate `fpem_CA_AC` in the i960 Cx/Hx/Jx processor's internal data RAM.

NOTE. *You cannot locate* `fpem_CA_AC` *into the data section of a PID program. You can allocate memory for* `fpem_CA_AC` *in the linker configuration file. To modify* `fpem_CA_AC`*, use the functions declared in the* `fpsl.h` *header file. The* `libmca.a` *and* `libhca.a` *libraries use* `fpem_CA_AC` *as an extension of the AC register; however, the* `libmstb.a` *library does not use* `fpem_CA_AC`.

Including the Header Files

To use a function defined in a library, you must include an external declaration of that function in your program. The header files contain declarations for the library functions and for variables and values that you can use with the library functions. Including header files can make developing a correct and efficient program easier, as follows:

- Some functions, such as those that accept `float` data types as arguments, require prototyped declarations. Since all function declarations in the header files are correctly prototyped, including the appropriate header files ensures that your use of a function matches the library definition of that function. You can write your own external declaration for any library function or variable, but doing so does not guarantee an exact match. The header files also define data types that exactly match the data types of function parameters and macros that provide convenient names for correct argument values.

- Some functions are also defined as macros or as inline assembly-language functions in the header files. Code resulting from a macro or inline assembly-language function expansion can execute more quickly and occupy less space than the code generated for a function call. Also, if you use a macro or assembly-language function, you need not link the library module containing the function.

To use the library function rather than the macro defined in an included header file, use `#undef` to remove the macro definition after defining the macro and before invoking the function. *C: A Reference Manual* describes how to define, use, and remove macros. As an alternative to removing the macro definition, you can disable macro expansion for the function identifier by putting parentheses around the function identifier in the function invocation. For example:

```
main()
{    (macro_name) (a);
}
```

You can include a header file in the same way as including any other source text file. The *i960 Processor Compiler User's Guide* explains how to use compiler options to include files.

Retargeting for Multi-tasking and Reentrancy

Low-level functions depend directly on the specific operation of the execution environment. The low-level libraries define functions for input/output (I/O), initialization, and cleanup specific to the MON960 debug monitor execution environments. You must rewrite these functions for execution in any other environment.

Additional low-level functions, such as thread and semaphore functions used in multi-tasking applications, are provided as stubs. An application involving multiple threads of execution can require that you implement the thread and semaphore functions. Chapter 5 explains how to rewrite the supplied low-level functions and how to implement new functions for multi-tasking and reentrant operation.

Since high-level functions are independent of the execution environment, you do not need to rewrite them. However, some high-level functions call low-level functions to perform I/O, initialization, and cleanup operations. If the high-level functions used in your program call low-level functions, you must rewrite the called low-level functions for your program to execute on any system other than those using the MON960 debug monitor. Chapter 5 explains the dependencies between specific high-level and low-level functions in the libraries. See Appendix A for a cross-reference list of low-level functions.

Identifying Run-time Errors

In addition to returning an error-indicator value, most library functions can set the value of the `errno` macro to provide more specific information about the cause of an error. The `errno` macro, defined in the `errno.h` header file, is specified by the ANSI standard to provide information about an error that has occurred.

The value of `errno` is useful when information about the most recent error is relevant. Once `errno` has been set because of an error, its value does not change until another error occurs. You can use `errno` effectively in the following ways:

- If a function can both set errno and return an error value, the return value of the function indicates whether an error occurred and the value of errno identifies the most recent error that has occurred.
- If a function can set errno but cannot return an error value, your program can identify an error occurring in the function as follows:
  - Set `errno` to `0` immediately before calling the function, so that `errno` does not contain a record of any previous error.

— Test `errno` immediately after the function returns. If `errno` is not `0`, an error has occurred in the function. The value of `errno` identifies the most recent error that has occurred.

The `errno.h` header file defines error macros that expand to the values used for `errno`. Include the `errno.h` header file via the `#include` directive.

Compiling for ANSI Compliance

You can use the `a` ic960 or `ansi` gcc960 compiler driver option to conditionally compile out all non-ANSI declarations and definitions from the ANSI-standard header files and to disable inline assembly-language functions and statements.

Header Files

3

The library header files contain source text declarations of library functions, variables, macros, and inline assembly functions. This chapter describes the non-ANSI header files and five of the ANSI header files which also contain compiler-specific information.

Chapter 4 of this supplement and Part II of *C: A Reference Manual* give more information on the operation and use of the individual ANSI functions and data types.

These ANSI library header files are described in *C: A Reference Manual*:

`assert.h`	Assertion evaluation.
`ctype.h`	Character testing and mapping.
`errno.h`	Error condition variables and macros.
`float.h`	Characteristics of floating-point types.
`limits.h`	Implementation limits.
`locale.h`	Localization. Although the `locale.h` header file declares functions and defines macros for localization, the libraries do not support localization.
`math.h`	Floating point math. Also described in this chapter.
`setjmp.h`	Non-local jumps.
`signal.h`	Signal and interrupt handling. Also described in this chapter.
`stdarg.h`	Variable arguments.

`stddef.h`	Standard language additions.
`stdio.h`	Stream input/output. Also described in this chapter.
`stdlib.h`	Utilities. Also described in this chapter.
`string.h`	String handling.
`time.h`	Date and time. Also described in this chapter.

These are the non-ANSI library header files described in this chapter:

`afpfault.h`	Accelerated floating-point library fault handling support. See Chapter 6 for information on fault handling support for the "libh" libraries.
`alloca.h`	Defines the `alloca` function.
`fcntl.h`	File access flag definitions.
`fpsl.h`	Floating-point operation control.
`__macros.h`	Defines macros for include files.
`reent.h`	Primitive functions for reentrant programming.
`search.h`	Linear search functions.
`stat.h`	File types and access permissions.
`std.h`	Standard system functions.
`types.h`	System V data-type definitions.
`unalign.h`	Defines special macros.
`varargs.h`	Defines macros for variable argument lists.

The following pages describe the non-ANSI header files and five ANSI header files (`math.h`, `signal.h`, `stdio.h`, `stdlib.h`, and `time.h`) which also contain compiler-specific information. These files are listed in alphabetical order by the names of the header files.

afpfault.h

Accelerated floating-point library fault handler.
non-ANSI

Discussion

This header file defines the interface to be used with the stub routines for fault handling provided in the AFP library (`libh`*`xx`*`.a`). The stub routines can be replaced in the library by user-defined routines as long as the interface defined in `afpfault.h` is used.

See Chapter 6 for a detailed discussion of floating-point library fault handling facilities.

alloca.h

Defines the `alloca` *function.*
non-ANSI

Discussion

The `alloca.h` header file declares the `alloca` function.

fcntl.h

File access flag definitions.
non-ANSI

Discussion

The `fcntl.h` header file defines macros for the flag values passed to the `open` function when opening a file. See Chapter 5 for a description of the `open` function.

The following macros set the access mode when you open a file:

`O_RDONLY`	Open a file in read-only mode.
`O_RDWR`	Open a file in read-write (update) mode.
`O_WRONLY`	Open a file in write-only mode.

The following macros set the file status for identifying and opening a file:

`O_APPEND`	Set the file pointer to the end of the file before each write operation.
`O_CREAT`	Create a new file.
`O_EXCL`	Use exclusive mode when opening the file.
`O_TRUNC`	Truncate the existing file's length to zero.

The following macros set the file type for the format of information to be read or written:

`O_BINARY`	Open a binary file.
`O_TEXT`	Open an ASCII file.

fpsl.h

Floating-point operation control.
non-ANSI

Discussion

The `fpsl.h` header file declares functions for controlling the i960 processor· floating-point operations and defines macros to be used as arguments to those functions. This header file also declares some non-ANSI math functions.

Use the following floating-point control functions, as described in Chapter 4, to read and modify parts of the arithmetic control (AC) register:

`fp_getround` `fp_setround`	read and modify the current rounding mode.
`fp_getmasks` `fp_setmasks`	read and modify the current exception masks.
`fp_getflags` `fp_setflags`	read and modify the current exception flags.
`fp_clrflags`	clears all the flags and returns the former flag values.
`fp_clriflag`	clears the interrupt overflow flag.
`fp_getenv` `fp_setenv`	read and modify the current floating-point environment.
`_getac` `_setac`	read and modify the entire AC register.

The following macros are valid arguments for the floating-point control functions. Use the following macros to read and write the floating-point exception flags:

`FPX_INVOP`	isolates the invalid-operation exception flag.
`FPX_ZDIV`	isolates the divide-by-zero exception flag.
`FPX_OVFL`	isolates the overflow exception flag.
`FPX_UNFL`	isolates the underflow exception flag.
`FPX_INEX`	isolates the inexact-result exception flag.
`FPX_CLEX`	clears all the exception flags.
`FPX_ALL`	sets all the exception flags.

Use the following macros to specify the rounding mode:

`FP_RN`	sets the rounding mode to round to nearest.
`FP_RM`	sets the rounding mode to round toward minus infinity.
`FP_RP`	sets the rounding mode to round toward plus infinity.
`FP_RZ`	sets the rounding mode to round toward zero (truncate).

The members of the _ac structure, defined in fpsl.h, isolate the fields of the AC register, as follows:

```
struct _ac {
  unsigned int        cc : 3; /* condition code       */
  unsigned int        as : 4; /* arithmetic status    */
  unsigned int           : 1;
  unsigned int iovfl_flg : 1; /* integer overflow flag */
  unsigned int           : 3;
  unsigned int iovfl_msk : 1; /* integer overflow mask */
  unsigned int           : 2;
  unsigned int    nif : 1; /* no-imprecise-faults flag */
  unsigned int fpflags : 5; /* fltg-pt-exception flags */
  unsigned int           : 3;
  unsigned int fpmasks : 5; /* fltg-pt-exception masks */
  unsigned int  nornmode : 1; /* normalizing mode     */
  unsigned int   rndmode : 2; /* rounding mode        */
};
```

The fpsl.h header file also declares non-ANSI functions. Function names ending with f, such as fp_logbf, take and return single-precision values. Function names ending with l, such as fp_logbl, take and return extended-precision values. The rest of the function names (e.g., fp_logb) take and return double-precision values.

The non-ANSI functions are:

fp_logbf fp_logb fp_logbl	return the base-2 logarithm.
fp_remf fp_rem fp_reml	return the remainder.
fp_rmdf fp_rmd fp_rmdl	return the remainder (IEEE).

`fp_roundf`
`fp_round`
`fp_roundl`
	round to an integral value.

`fp_scalef`
`fp_scale`
`fp_scalel`
	perform a scaling operation.

__macros.h

Defines macros for include files.
non-ANSI

Discussion

The `__macros.h` header file defines macros used by the other include files. These macros are defined for portability of the system include files, and are subject to change with each compiler release.

math.h

Floating-point math.
ANSI

Discussion

The `math.h` header file declares both ANSI-standard and i960-specific floating-point arithmetic functions. The ANSI-standard part of `math.h` is described in *C: A Reference Manual*.

The ANSI-standard mathematics functions are declared as double-precision floating-point functions for all i960 processors. The following mathematics functions are also available as single-precision floating-point functions on all i960 processors:

`atanf`	`expf`	`powf`
`atan2f`	`floorf`	`sinf`
`ceilf`	`fabsf`	`sqrtf`
`cosf`	`logf`	`_IEEE_sqrtf`
		`tanf`

NOTE. *There are two implementations of* `sqrt` *for each precision. The* `_IEEE_sqrt` *and* `_IEEE_sqrtf` *functions are fully IEEE-754 conformant in that they perform fault checking as specified in the IEEE-754 specification. The ANSI versions,* `sqrt` *and* `sqrtf`*, unconditionally set* `errno` *to* `EDOM` *when given inappropriate values.*

The following single-precision versions of ANSI-standard floating-point functions are available for i960 processors with on-chip floating-point support:

`acosf`	`log10f`
`asinf`	

If you do not specify the `-a` (ic960) or `-ansi` (gcc960) option when compiling, `math.h` declares the following non-ANSI functions in addition to the standard functions:

`square` returns the square of a number.

`hypot` returns the hypotenuse.

If you do not specify the `-a` or `-ansi` (ANSI) option (`-a` for ic960, `-ansi` for gcc960) when compiling and the i960 processor is without on-chip floating-point support, `math.h` declares the following non-ANSI functions in addition to the standard functions:

`_IEEE_sqrt`	double precision
`_IEEE_sqrtf`	single precision

If you do not specify the `-a` or `-ansi` (ANSI) option (`-a` for ic960, `-ansi` for gcc960) when compiling, the `math.h` header file also defines the following structure data type for handling complex numbers:

```
struct complex { double x, y };
```

See Chapter 4 for a description of the `_IEEE_sqrtf`, `hypot`, and `square` functions.

reent.h

Primitive functions for reentrant programming.
non-ANSI

Discussion

The `reent.h` header file declares the low-level input/output (I/O) and thread functions used for reentrant programming. Many portable functions in the libraries call these low-level functions.

Since low-level functions interact directly with the execution environment, you must rewrite them to conform to your execution environment, as described in Chapter 5.

search.h

Linear search functions.
non-ANSI

Discussion

The `search.h` header file declares the linear search functions `lfind` and `lsearch`. Use `lfind` and `lsearch` to find items in an unsorted list, as described in Chapter 4.

signal.h

Signal and interrupt handling.
ANSI

Discussion

Both the ANSI and POSIX standards describe signals as conditions that can be reported asynchronously during program execution. The `signal.h` header file provides declarations and definitions for handling ANSI and POSIX signals. The ANSI signal-handling functions and macros are described in *C: A Reference Manual*. The non-ANSI signal macros defined in `signal.h` are:

`SIGREAD`	indicates that a physical read operation has returned an end-of-file value.
`SIGWRITE`	indicates that a write operation has failed.
`SIGALLOC`	indicates that memory allocation has failed.

`SIGFREE`	indicates that an invalid pointer argument has been passed to a deallocation function.
`SIGUSR1`	is user-defined.
`SIGUSR2`	is user-defined.
`SIGSIZE`	indicates the number of defined signals.

stat.h

File types and access permissions.
POSIX

Discussion

The `stat.h` header file defines macros used as masks to check and set the type and access permissions of files on the host system supporting the execution vehicle. The `stat.h` header file also declares the `fstat` and `stat` functions, described in Chapter 5, and the structure `stat`, used as an argument to `fstat` and `stat`.

Additional status and file-type macros defined in `stat.h` are available for UNIX compatibility and are not supported on Windows.

std.h

System functions.
non-ANSI

Discussion

The `std.h` header file declares operating system functions.

stdio.h

Stream input/output.
ANSI

Discussion

The `stdio.h` header file declares functions for stream input and output (I/O). The ANSI part of `stdio.h` is described in *C: A Reference Manual*. In addition, if you do not specify the `-a` or `-ansi` (ANSI) option (`-a` for ic960, `-ansi` for gcc960) when compiling, `stdio.h` defines the following non-ANSI functions:

`fcloseall`	closes all open files.
`fdopen`	opens a file.
`fgetchar`	reads a character.
`fileno`	gets the file descriptor for a stream.
`flushall`	empties all input and output buffers.
`fputchar`	writes a character.
`getw`	reads a word.

`putw`	writes a word.
`rmtmp`	removes a temporary file.

See Chapter 4 for a detailed description of the use of each function.

stdlib.h

Utilities.
ANSI

Discussion

The `stdlib.h` header file declares general utility functions. The ANSI contents of `stdlib.h` are described in *C: A Reference Manual*. In addition, if you do not specify the `-a` or `-ansi` (ANSI) option (`-a` for ic960, `-ansi` for gcc960), `stdlib.h` defines the following non-ANSI functions:

`ecvt`, `fcvt`, `gcvt`	convert a floating-point number to a string.
`getopt`	returns the next letter in the argument that matches a letter in the string argument.
`itoa`	converts an integer to a string.
`itoh`	converts an integer to hexadecimal.
`ltoa`, `ltos`	convert a long integer to a string.
`ltoh`	converts a long integer to hexadecimal.
`ultoa`	converts an unsigned long integer to a string.
`utoa`	converts an unsigned integer to a string.

See Chapter 4 for a detailed description of each function.

string.h

Character array manipulation.
ANSI

Discussion

The `string.h` header file declares functions for manipulating character arrays. The ANSI contents of `string.h` are described in *C: A Reference Manual*. In addition, if you do not specify the `-a` or `-ansi` (ANSI) option (`-a` for ic960, `-ansi` for gcc960), `string.h` defines the following non-ANSI functions:

`memicmp`	compares two strings in memory, ignoring case.
`strdup`	duplicates a string.
`stricmp` `strnicmp`	compare two strings, ignoring distinctions between uppercase and lowercase.
`strlwr` `strupr`	convert a string to lowercase or to uppercase, respectively.
`strnset` `strset`	assign values to characters in a string.
`strrev`	reverses the order of characters in a string.

See Chapter 4 for a detailed description of each function.

time.h

Date and time.
ANSI

Discussion

The `time.h` header file provides functions and macros for determining the current time, elapsed time, and timezone. The non-ANSI time functions are described in Chapter 4. The ANSI-standard part of `time.h` is described in *C: A Reference Manual*. If you do not specify the `-a` or `-ansi` (ANSI) option (`-a` for ic960, `-ansi` for gcc960), `time.h` also defines the following:

`daylight` macro	indicates whether daylight savings time is in effect.
`timezone` macro	provides the difference in seconds between Coordinated Universal Time and local time.
`tzname` macro	provides a pair of strings that identify the name of the time zone and the name of the daylight savings time.
`tzset` function	sets the values of `daylight`, `timezone`, and `tzname`.

See the `tzset` entry in Chapter 4 for a description of these facilities.

types.h

System V data-type definitions.
non-ANSI

Discussion

The `types.h` header file defines the following data types used for compatibility with UNIX System V:

`uchar` `u_char`	are the same as `unsigned char`.
`ushort` `u_short`	are the same as `unsigned short`.
`uint` `u_int`	are the same as `unsigned int`.
`ulong` `u_long`	are the same as `unsigned long`.
`dev_t`	is the same as `short`. The `stat` structure uses this data type to identify a device.
`off_t`	is the same as `long`. The `stat` structure uses this data type to contain a file size in bytes.
`mode_t`	is the same as `unsigned long`.
`size_t`	is the same as `unsigned`.

unalign.h

Defines special macros.
non-ANSI

Discussion

This include file defines special macros for accessing 16-bit short and 32-bit word-length quantities on unaligned addresses. Unaligned accesses are faster with the i960 CA processor using the compiler-scheduled instructions than allowing the microcode and/or bus controller to handle them.

The macros defined are:

```
GET_UNALIGNED_WORD
SET_UNALIGNED_WORD
```

For word accesses which are unaligned more than 10% of the time, and the alignment is not always 2-byte.

```
GET_UNALIGNED2_WORD
SET_UNALIGNED2_WORD
```

For word accesses which are unaligned more than 10% of the time and the alignment is always 2-byte.

```
GET_UNALIGNED_SHORT
SET_UNALIGNED_SHORT
```

For signed short accesses which are unaligned more than 10% of the time.

```
GET_UNALIGNED_UNSIGNED_SHORT
SET_UNALIGNED_UNSIGNED_SHORT
```

For unsigned short accesses which are unaligned more than 10% of the time.

Use standard C syntax for naturally aligned data references (structure fields not under `#pragma pack` or `#pragma align` and pointer dereferences without a cast). The macros in this file provide a method of abstracting non-natural data references so that the application does not have to concern itself with how unaligned accesses are performed.

By default, the macros are generated for unaligned accesses in little-endian memory regions. If the preprocessor symbol `__i960_BIG_ENDIAN__` is defined, the macros are generated for big-endian memory accesses. The compiler option `-G` defines `__i960_BIG_ENDIAN__`.

If you are a big-endian memory user using an i960 CA processor D-step (or later) part, the chip supports unaligned accesses in big-endian memory regions. Earlier (pre-D-step) parts will fault on any unaligned accesses in big-endian memory regions.

Therefore, if you have a pre-D-step part and there is a possibility that a memory access will be unaligned, you must use one of the `UNALIGNED` or `UNALIGNED2` macros above or you will get a fault.

varargs.h

Defines macros for variable argument lists. non-ANSI

Discussion

The `varargs.h` header file defines macros that provide a means of writing procedures that accept variable argument lists and which are portable to pre-ANSI C environments.

The macros defined are:

`va_alist`	is used in a function header to declare a variable argument list.
`va_arg`	returns the next argument in the list pointed to by its parameters.
`va_dcl`	is a declaration for `va_alist`.
`va_end`	is used to finish up.
`va_start`	is called to initialize parameters to the beginning of the list.

See *C: A Reference Manual* for a discussion of these facilities

Header File Changes to Support the Long Long Type

The following header files have been changed to support the *long long* type:

`limits.h` `stdlib.h`

The header file `limits.h` includes definitions for the following pre-processor constants:

`LLONG_MAX`	`Maximum value for an object of type long long int`
`LLONG_MIN`	`Minimum value for an object of type long long int`
`ULLONG_MAX`	`Maximum value for an object of type unsigned long long int`

The header file `stdlib.h` includes prototypes for the following functions:

```
atoll      strtoll     strtoull     lltoa    lltoh
lltos      ulltoa
```

The 80960 ABI currently does not define support for the *long long* type. To prevent compile time warnings when compiling with the abi (ic960 option `Gabi` or gcc960 option `mabi`) compiler switch or the ANSI (ic960 option `a` or gcc960 option `ansi`) compiler switch, the above definitions are included conditionally only when the macros `__STRICT_ANSI__`, `__STRICT_ANSI`, and `__i960_ABI__` are not defined. If you would like to include the above definitions even when using the abi or the ANSI compiler switches, please define the pre-processor macro `__USE_LONG_LONG__` by using the compiler option `-D__USE_LONG_LONG__`.

Please note that the prototypes for the following functions are not included when compiling with the ANSI switch even when the macro `__USE_LONG_LONG__` is defined:

```
lltoa      lltoh      lltos      ulltoa
```

Library Functions 4

This chapter describes the library functions that are not fully described in *C: A Reference Manual*.

These functions are portable and you need not rewrite them to retarget your application program. However, some of these functions can call primitive functions that must be rewritten for any execution environment not supported by the Intel MON960 debug monitor. Retargeting is described in Chapter 5. See Appendix A for a cross-reference list of the primitive functions.

ecvt, fcvt, gcvt

Convert floating-point number to string.

```
char *ecvt (double value, int count, int *dec, int *sign);
char *fcvt (double value, int count, int *dec, int *sign);
char *gcvt (double value, int count, char *buffer);
```

value	is the floating-point number to be converted.
count	is the desired number of digits in the converted string, excluding the terminating null character.
dec	is a pointer to a variable containing the implied position of the decimal point in the converted string.

sign	is a pointer to a variable containing the sign of the floating-point value.
buffer	is a pointer to a buffer for the converted string.
Header File	`stdlib.h`

Discussion

Use `ecvt`, `fcvt`, or `gcvt` to convert *value* to a null-terminated character string. The converted string contains only digits and the terminating null character. The `gcvt` function stores the string at the location pointed to by *buffer*.

The *count* argument specifies how many digits are stored after the implied decimal point. If the conversion produces more than *count* digits, the low-order digit is rounded. If *count* is larger than the number of digits, the string is padded with zeros to fill the specified length. For `gcvt`, the buffer must be large enough to hold the converted string and terminating null character.

If possible, `gcvt` formats the string in the decimal (`%f`) format used by the `printf` function; otherwise, `gcvt` formats the string in the exponential (`%e`) format. You use also `ecvt` to format the string in the exponential format used by `printf` or `fcvt` to format the string in decimal format.

The converted string contains only digits. To find the position of the implied decimal point and sign, use *dec* and *sign* after the function call. The *dec* argument points to an integer that indicates the decimal position relative to the beginning of the string. A negative or zero value indicates a position preceding the first digit in the string. The *sign* argument points to an integer that indicates the sign of the floating-point string. The integer is zero for a positive value and non-zero for a negative value.

NOTE. *The* `ecvt`, `fcvt`, *and* `gcvt` *functions are not reentrant. Use the* `sprintf` *function, described in* C: A Reference Manual, *instead for portability.*

Returns

The `ecvt`, `fcvt`, and `gcvt` functions return a pointer to the converted string. These functions do not return any special value to indicate an error.

Related Topic

`sprintf` (*C: A Reference Manual*)

fcloseall

Close all open streams.

```
int fcloseall (void);
```

Header File `stdio.h`

Discussion

Use this function to close all currently open files. The `fcloseall` function, however, does not close `stdin`, `stdout`, or `stderr`.

Returns

The `fcloseall` function returns the number of files closed, which can be zero or greater. This function does not return any special value to indicate an error.

Related Topics

`fopen` (*C: A Reference Manual*)
`stderr` (*C: A Reference Manual*)
`stdin` (*C: A Reference Manual*)
`stdout` (*C: A Reference Manual*)

fdopen

Open a stream with a file descriptor.
POSIX 8.2.2

```
FILE *fdopen (int fildes, char *mode);
```

fildes	is the file descriptor.
mode	is one of the file opening modes used by the `fopen` function described in the *C: A Reference Manual*, except that the `w` and `w+` modes do not cause truncation of the file.
Header File	`stdio.h`

Discussion

Use this function to open a stream and associate it with the file descriptor *fildes*. The file to be associated with *fildes* must already be open.

You cannot open a stream in a mode incompatible with the mode of the file. For example, if the file is open for writing, you cannot open the stream for reading or for updating.

Returns

On successful completion, `fdopen` returns a pointer to the stream; otherwise `fdopen` returns a `NULL` pointer, which indicates an invalid file *mode*.

Related Topics

`fcntl.h` (Chapter 3)
`fopen` (*C: A Reference Manual*)
`open` (*C: A Reference Manual*)

fgetchar

Read character from standard input stream.

```
int fgetchar (void);
```

Header File `stdio.h`

Discussion

Use this function to read a character from the standard input stream, `stdin`. For example, the following program uses `fgetchar` to echo the input to the screen, one character at a time:

```
#include <stdio.h>
main()
{
  int ch;
  fputs("Enter Data Terminated by EOF >", stdout):
  while ((ch = fgetchar()) != EOF)
    fputc (ch, stdout);
}
```

Returns

On successful completion, `fgetchar` returns the next character from `stdin`; otherwise, `fgetchar` returns `EOF`. Since `EOF` is a legal `int` value, use the `feof` or `ferror` function, described in *C: A Reference Manual*, to check for an actual error.

Related Topics

`feof`	(*C: A Reference Manual*)
`ferror`	(*C: A Reference Manual*)
`stdin`	(*C: A Reference Manual*)

fileno

Get file descriptor for stream. POSIX 8.2.1

`int fileno (FILE *stream);`

stream	is a pointer to an open stream.
Header File	`stdio.h`

Discussion

Use this function to get the file descriptor associated with the given *stream*. This function lets you use the file-descriptor I/O calls (for example, `read`, `write`, and `lseek`) on streams.

To mix the two I/O systems, such as `open` vs. `fopen`, you must flush all I/O buffers when going from the buffered system to the unbuffered system. If you omit this step, you can lose data.

Returns

On successful completion, `fileno` returns the file descriptor. This function does not return any special value to indicate an error.

Related Topics

`fdopen`		`open`	(Chapter 5)
`fopen`	(*C: A Reference Manual*)	`read`	(Chapter 5)
`lseek`	(Chapter 5)	`write`	(Chapter 5)

flushall

Flush all streams.

```
int flushall (void);
```

Header File `stdio.h`

Discussion

Use this function to write output stream buffers to the associated files and clear open input streams of their contents. The `flushall` function does not close the streams.

Returns

The `flushall` function returns the number of streams successfully flushed. This function does not return any special value to indicate an error.

fputchar

Write a character to standard output stream.

```
int fputchar (int c);
```

c is the character to be written.

Header File `stdio.h`

Discussion

Use this function to write a character to `stdout`. The `fputchar` function is the same as `fputc(c,stdout)`. For example, the following program uses the `fputchar` function to echo console input to the screen one character at a time:

```
#include <stdio.h>
main()
{
  int ch;
  fputs("Enter Data Terminated by EOF  ",stdout);
  while((ch=fgetchar()) != EOF)
    fputchar(ch);
}
```

Returns

On successful completion, `fputchar` returns the character written; otherwise, `fputchar` returns `EOF`. Since `EOF` is a legal `int` value, use the `ferror` function, described in *C: A Reference Manual*, to check for an actual error.

Related Topics

`ferror`	(*C: A Reference Manual*)
`fgetchar`	
`fputc`	(*C: A Reference Manual*)

fp_getenv, fp_setenv

Read and modify arithmetic controls (i960 processor-specific).

```
unsigned fp_getenv (void);
unsigned fp_setenv (unsigned val);
```

val	is the bit pattern for setting the arithmetic controls.
Header File	`fpsl.h`

Discussion

Use `fp_getenv` to read the floating-point bits of the arithmetic controls (AC) register. Use `fp_setenv` to set the floating-point bits of the AC register. For example, the following statement sets the rounding mode for round-to-nearest, sets normalizing mode on, masks all exceptions other than the invalid-operation exception, and clears all exception flags:

```
(void) fp_setenv(0x3b000000);
```

For more information on the AC register, see your assembler user's guide.

Returns

On successful completion, `fp_getenv` returns the current AC register contents and `fp_setenv` returns the previous AC register contents. These functions do not return any special value to indicate an error.

fp_getflags, fp_setflags, fp_clrflags, fp_clriflag

Read and modify floating-point exception flags (i960 processor-specific).

```
int fp_getflags (void);
int fp_setflags (int val);
int fp_clrflags (int val);
int fp_clriflag (void);
```

`val` is the bit pattern for setting the exception flags.

Header File `fpsl.h`

Discussion

Use `fp_getflags` to read the current exception flags from the floating-point AC register. Use `fp_setflags` to set any of the exception flags to 1 and `fp_clrflags` to clear any of the exception flags to zero. Use `fp_clriflag` to clear the interrupt overflow flag. The `fp_setflags` and `fp_clrflags` functions also return the previous values of all the exception flags. For example, the following statement fetches the exception flags into the `fpex_flags` variable:

```
fpex_flags = fp_getflags();
```

The `fp_setflags` and `fp_clrflags` functions use only the 5 low-order bits of *`val`*. To operate on any particular flag, set the corresponding bit in *`val`* to `1` as follows:

- Set val bit 0 to change the overflow flag (bit 16 of the AC register).
- Set val bit 1 to change the underflow flag (bit 17 of the AC register).
- Set val bit 2 to change the invalid-operation flag (bit 18 of the AC register).

- Set val bit 3 to change the zero-divide flag (bit 19 of the AC register).
- Set val bit 4 to change the inexact flag (bit 20 of the AC register).

Returns

On successful completion, `fp_getflags` returns the current exception flags values. The `fp_setflags`, `fp_clrflags`, and `fp_clriflag` functions return the previous flag values. These functions do not return any special value to indicate an error.

Related Topics

`fpgetenv, fp_setenv`

fp_getmasks, fp_setmasks

Read and modify floating-point exception masks (i960 processor-specific).

```
int fp_getmasks (void);
int fp_setmasks (int val);
```

`val` is the bit pattern for setting the exception masks.

Header File `fpsl.h`

Discussion

Use `fp_getmasks` to read the current exception mask bits from the floating-point AC register. Use `fp_setmasks` to set any of the exception mask bits to a specified value. For example, the following statement masks the invalid-operation exception:

```
(void) fp_setmasks(0x04);
```

The `fp_setmasks` function uses only the 5 low-order bits of *`val`*. To operate on any particular mask bit, set the corresponding bit in *`val`* as follows:

- Set val bit 0 to change the overflow mask (bit 24 of the AC register).
- Set val bit 1 to change the underflow mask (bit 25 of the AC register).
- Set val bit 2 to change the invalid-operation mask (bit 26 of the AC register).
- Set val bit 3 to change the zero-divide mask (bit 27 of the AC register).
- Set val bit 4 to change the inexact mask (bit 28 of the AC register).

Returns

On successful completion, `fp_getmasks` returns the current mask values and `fp_setmasks` returns the previous values. These functions do not return any special value to indicate an error.

Related Topics

`fpgetenv`, `fp_setenv`

fp_getround, fp_setround

Read and modify floating-point rounding mode (i960 processor-specific).

```
int fp_getround (void);
int fp_setround (int val);
```

val is the bit pattern for setting the rounding mode.

Header File fpsl.h

Discussion

Use `fp_getround` to read the current rounding mode from the floating-point AC register. Use `fp_setround` to set the rounding mode to a specified value. The `fp_setround` function also returns the previous value of the rounding mode. For example, the following statement sets the rounding mode to truncate and saves the previous rounding mode in the `save_rm` variable:

```
save_rm = fp_setround(3);
```

These functions use only the two low-order bits of *val*, forcing the rounding mode value to be in the range `0` to `3`. To specify a rounding mode, you can use the following values for *val*:

- Use 0 to specify round-to-nearest.
- Use 1 to specify rounding down (toward minus infinity).
- Use 2 to specify rounding up (toward plus infinity).
- Use 3 to specify truncation (toward 0).

Returns

On successful completion, `fp_getround` returns the current rounding mode and `fp_setround` returns the previous rounding mode. These functions do not return any special value to indicate an error.

Related Topics

`fpgetenv, fp_setenv`

_getac, _setac

Read and modify arithmetic controls (i960 processor-specific).

```
unsigned _getac (void);
unsigned _setac (unsigned val);
```

val	is the bit pattern for setting the arithmetic controls.
Header File	`fpsl.h`

Discussion

Use `_getac` to read the current value of the arithmetic controls (AC) register. Use `_setac` to set the AC register. The `_setac` function also returns the previous value of the AC register. For example, the following statement sets the arithmetic controls correctly for the C run-time library functions, including the integer overflow fault, floating-point overflow fault, floating-point underflow fault, floating-point zero-divide fault, floating-point inexact fault, denormalized numbers, and round-to-nearest rounding mode:

```
old_ac = _setac(0x3b001000)
```

You can use `_getac` and `_setac` on any i960 processor even though the i960 CA processor uses the `fpem_CA_AC` external variable. The `libmxx` floating-point library for each processor contains an appropriate implementation of these functions.

The operation of `_getac` and `_setac` on each processor is as follows:

- On the i960 CA and CF processors, _getac returns the value of the AC register ORed with fpem_CA_AC. The _setac function sets both the AC register and fpem_CA_AC.
- On other i960 processors, _getac and _setac return and set the AC register value, respectively.

Returns

On completion, `_getac` returns the value of the AC register or the `fpem_CA_AC` variable. The `_setac` function returns the previous value of the AC register or the `fpem_CA_AC` variable. These functions do not return any special value to indicate an error.

Related Topics

`fp_getenv, fp_setenv`

getw

Read integer from stream. SVID

```
int getw (FILE *stream);
```

`stream` identifies the input stream.

Header File `stdio.h`

Discussion

Use this function to read the next two bytes from the stream opened by `fopen` or `creat`. The apparent behavior of this function can vary due to word length and byte ordering in the environment in which the stream is written using `putw`. For example, the following program copies the binary file `filename.in` to the file `filename.out`:

```
#include <stdio.h>

main()
{
  FILE *instream, *outstream;

  int word;

  if (!(instream = fopen("filename.in", "rb")))
     return;

  if (!(outstream = fopen("filename.out", "wb")))
  {  fclose(instream);
     return;
  }

  while ((word = getw(instream)) != EOF)
     putw(word, outstream);

  fclose(outstream);
  fclose(instream);
}
```

Returns

On successful completion, `getw` returns the input word; otherwise, `getw` returns `EOF` as an error or end-of-file indicator.

Since the error and end-of-file indicators are both `EOF`, which can also be a valid data word, use `feof` and `ferror` to distinguish between end-of-file, an error, or a valid return of `EOF`.

Related Topics

`creat`	(Chapter 5)	`fopen`	(*C: A Reference Manual*)
`feof`	(*C: A Reference Manual*)	`putw`	
`ferror`	(*C: A Reference Manual*)		

getopt

Get option letter from argument vector.

```
int (getopt)(int argc, char **argv, char *optstring);
```

`argc` the number of pointers in *`argv`*.

`argv` points to the index of the next command line argument to be processed.

`optstring` points to the string containing the option letters.

Header File `stdlib.h`

Discussion

Function `getopt` returns the next option letter in `argv` that matches a letter in *`optstring`*. *`optstring`* must contain the option letters recognized by the command line command using `getopt()`. If a letter is followed by a colon, the option is expected to have an argument or group of arguments which must be separated from it by white space.

`optarg` is set to point to the start of the option argument on return from `getopt`.

`getopt` places the argv index of the next argument to be processed in `optind`. The external function `optind()` is initialized to 1 before the first call to `getopt`.

When all options have been processed (up to the first non-option argument) `getopt` returns -1. The special option `"--"` can be used to delimit the end of the options; when it is encountered, -1 is returned and `"--"` is skipped.

Returns

This function returns the next option letter in *`argv`* that matches a letter in *`optstring`*.

hypot

Find the Euclidean distance.

```
double hypot (double x, double y);
```

x and *y*	are double-precision floating-point values.
Header File	`math.h`

Discussion

Use this function to find and return the hypotenuse for sides of lengths *x* and *y*, that is, the square root of the sum of the squares of *x* and *y*.

Returns

$\sqrt{(x2 + y2)}$

_IEEE_sqrt, _IEEE_sqrtf

Determine the IEEE conformant square root of a value.

```
double _IEEE_sqrt(double x);
float _IEEE_sqrtf(float x);
```

x is a user provided value.

Header File `math.h`

Discussion

The `_IEEE_sqrt` and `_IEEE_sqrtf` functions produce the square root of the value provided in `x`. The `_IEEE_sqrt` functions conform fully to IEEE-754. `_IEEE_sqrtf` provides single precision accurracy. `_IEEE_sqrt` provides double precision accurracy.

Return Value

Upon successful completion, `_IEEE_sqrtf` returns the single precision square root of the value in *x*. The function performs fault checking in conformance with the IEEE-754 specification.

Upon successful completion, `_IEEE_sqrt` returns the double precision square root of the value in *x*. The function performs fault checking in conformance with the IEEE-754 specification.

itoa

Convert integer to string.

```
char *itoa (int value, char *string, int radix);
```

value	is the integer to be converted.
string	is a pointer to the string.
radix	is the radix of *value*, in the range 2 through 36.
Header File	stdlib.h

Discussion

Use this function to convert the input integer *value* to the equivalent null-terminated character string and store the result in *string*. Specify the sign of *value* and the base of the conversion with the *radix* argument. The absolute value of *radix* must be in the range 2 through 36. If *radix* is negative, *value* is interpreted as signed. If *radix* is positive, *value* is interpreted as unsigned. For example, the following program converts the number in *value* to a decimal ASCII string in the *string* variable and prints the value of *string*:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
  int value;
  char string[34];
  char * num;

  value = 12;
  num = itoa(value, string, 10);
  printf("%s\n", string);
}
```

The *string* buffer must be large enough to hold the ASCII representation of the largest integer possible in your execution environment.

Returns

The `itoa` function returns a pointer to the string. This function does not return any special value to indicate an error.

Related Topic

`sprintf` (*C: A Reference Manual*)

itoh

Convert integer to hexadecimal.

```
char *itoh (int n, char *buffer);
```

n	is the integer to be converted.
buffer	is a pointer to the string.
Header File	`stdlib.h`

Discussion

Use this function to convert the input integer *n* into the equivalent null-terminated hexadecimal string in the buffer pointed to by *buffer*. The buffer must be large enough to hold the hexadecimal representation of the largest integer possible in your execution environment. This function converts all hexadecimal characters to lowercase. For example, the following program converts the number in the variable `n` to a hexadecimal ASCII string in `hexstr` and prints the `hexstr`:

```
#include <stdlib.h
#include <stdio.h
```

```
main()
{
  unsigned int n;
  char hexstr[9];
  char * number;

  n = 0x3ff;
  number = itoh(n, hexstr);
  printf("%s\n", hexstr);
}
```

For portability, use `sprintf` with the `%x` conversion specifier.

Returns

The `itoh` function returns a pointer to the string. This function does not return any special value to indicate an error.

Related Topic

`sprintf` (*C: A Reference Manual*)

lfind, lsearch

lfind - Linear search
lsearch - Linear search and update.
SVID

```
char *lfind (const char *key, const char *base,
          unsigned *nelp, unsigned width,
          int (*compar)(const void *, const void *));

char *lsearch (const char *key, char *base,
          unsigned *nelp, unsigned width,
          int (*compar)(const void *, const void *));
```

`key` is a pointer to the value to be searched for.

base	is a pointer to the first element in the array.
nelp	is a pointer to the number of elements in the array.
width	is the size, in bytes, of each element in the array.
compar	points to the function to compare each element in the array with the *key*.
Header File	`search.h`

Discussion

Use `lfind` or `lsearch` to perform a linear search of an array of elements beginning at *base* and searching to the first occurrence of *key*. The value of *nelp* points to the number of elements in the array. *width* indicates the size of each element in bytes. The array need not be sorted.

If `lsearch` does not find a match, it adds *key* to the end of the array, and returns a pointer to the new position of *key*. Since `lsearch` does not allocate space for a new element, you must ensure that space is available for the element.

You must supply the comparison function that *compar* points to. The comparison function must take two arguments pointing to the elements to be compared, return `0` if the elements are identical, and return non-zero otherwise.

Returns

Both functions return a pointer to the first match. If `lfind` does not find a match, it returns a `NULL` pointer. If `lsearch` appends *key* to the array, the return value is a pointer to the new *key* element in the array. These functions do not return any special value to indicate an error.

Related Topic

`bsearch` (*C: A Reference Manual*)

ltoa, ltos

Convert long integer to string.

`char *ltoa (long` *`num`*`, char *`*`string`*`, int` *`radix`*`);`

`char *ltos (long` *`num`*`, char *`*`string`*`, int` *`radix`*`);`

`num`	is the integer to be converted.
`string`	is the pointer to the string.
`radix`	is the radix of *`num`*, in the range `2` through `36` decimal.
Header File	`stdlib.h`

Discussion

Use `ltoa` to convert the supplied `long int` value in *`num`* to the equivalent ASCII string in the *`string`* buffer using base *`radix`*, which must be in the range `2` through `36` decimal. For example, the following program uses `ltoa` to convert a number in the variable `number` to an ASCII string in the variable `longstr` and prints the `longstr` string:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
  long number;
  char longstr[12];
  char * buf;

  number = 10223444L;
  buf = ltoa(number, longstr, 10);
   /* longstr contains "10223444" */
  printf("%s\n", longstr);
}
```

For `ltos`, *`radix`* can be an integer value from `2` to `36` or `-2` to `-36` decimal. The absolute value of *`radix`* is the number base of the input argument. A negative *`radix`* indicates that the input value is a `signed long`. A positive radix indicates an `unsigned long` input.

The buffer must be large enough to hold the largest number possible in your execution environment. The string is null-terminated.

Returns

The `ltoa` and `ltos` functions return a pointer to the string. This function does not return any special value to indicate an error.

Related Topics

`ltoh`
`ultoa`, `utoa`

ltoh

Convert long integer to hexadecimal

```
char *ltoh (unsigned long num, char *string);
```

`num`	is the integer to be converted.
`string`	is the pointer to the string.
Header File	`stdlib.h`

Discussion

Use this function to convert the `long integer` value in *`num`* into the equivalent hexadecimal string in the *`string`* buffer. The buffer must be large enough to hold the hexadecimal representation of the largest possible integer. For example, the following program uses `ltoh` to convert the number in the variable `number` to an ASCII value in the variable `hexstr` and prints the `hexstr` string:

```
#include <stdlib.h>
#include <stdio.h>

main()
{
  unsigned long number;
  char hexstr[9];
  char * buf;

  number = 10223444L;
  buf = ltoh(number, hexstr);
  /* hexstr contains "9BFF54" */
  printf("%s\n", hexstr);
}
```

For portability, use `sprintf` with the `%lx` conversion specifier.

Returns

The `ltoh` function returns a pointer to the string. This function does not return any special value to indicate an error.

Related Topics

`ltoa`, `ltos`
`sprintf` (*C: A Reference Manual*)
`ultoa`, `utoa`

memicmp

Compare characters in memory, ignore case.

```
int memicmp (const void *ptr1, const void *ptr2,
    unsigned len);
```

ptr1	points to the source string.
ptr2	points to the destination string.
len	is the number of characters to compare.
Header File	`string.h`

Discussion

Use this function to compare two strings lexicographically, ignoring differences between lowercase and uppercase. The `memicmp` function is a case-insensitive version of the ANSI function `memcmp`. As such, `memicmp` compares *len* characters, starting at *ptr1*, with *len* characters at *ptr2*. The result indicates whether the first string is less than, equal to, or greater than the second string, ignoring the case of each string. The digits in the strings are compared lexicographically; that is, as characters and not as values. For example, `2` is greater than `13`, but `02` is less than `13`.

Returns

If the first string is lexicographically less than the second (ignoring case), `memicmp` returns a negative integer. If the first string is greater (ignoring case), `memicmp` returns a positive integer. If the strings are equal, `memicmp` returns `0`. This function does not return any special value to indicate an error.

Related Topics

`memcmp`	(*C: A Reference Manual*)
`stricmp`	
`strnicmp`	

putw

Write integer to stream.
SVID

```
int putw (int w, FILE *stream);
```

`w`	contains the two bytes to be written.
`stream`	Identifies the output stream.
Header File	`stdio.h`

Discussion

Use this function to write *`w`* to the specified stream. This function writes the least-significant byte of the word first.

Returns

On successful completion, `putw` returns the word written, which can be `EOF`. You can use `feof` and `ferror` to distinguish between an error and a valid return of `EOF`.

Related Topics

`feof`	(*C: A Reference Manual*)
`ferror`	(*C: A Reference Manual*)
`getw`	

rmtmp

Remove temporary files.

```
int rmtmp (void);
```

Header File `stdio.h`

Discussion

Use this function to close and delete any files opened by the function `tmpfile`, described in *C: A Reference Manual*.

Returns

The `rmtmp` function returns the number of files deleted. This function does not return any special value to indicate an error.

Related Topic

`tmpfile` (*C: A Reference Manual*)

square

Square a number.

```
double square (double val);
```

`val` is the number to be squared.

Header File `math.h`

Discussion

Use this function to calculate the square of the number *val* (that is, *val* * *val*).

Returns

The `square` function returns the value of *val* * *val*. This function does not return any special value to indicate an error.

strdup

Duplicate string.

```
char *strdup (const char *s);
```

s	points to a character string to be copied.
Header File	`string.h`

Discussion

Use this function to copy the character string pointed to by *s*. The `malloc` function is called to obtain the memory space needed for the copy. Use `free` to return the memory space when the program no longer needs it.

Returns

The `strdup` function returns a pointer to the duplicate string placed in memory. This function returns `NULL` if `malloc` cannot allocate the required memory.

Related Topics

`free`	(*C: A Reference Manual*)
`malloc`	(*C: A Reference Manual*)

stricmp

Compare strings, ignore case.

```
int stricmp (const char *s1, const char *s2);
```

s1, *s2* point to the strings to be compared.

Header File `string.h`

Discussion

Use this function to compare two strings lexicographically, ignoring distinctions between lowercase and uppercase. The `stricmp` function is a case-insensitive version of the ANSI `strcmp` function. As such, `stricmp` compares the first null-terminated string to the second and returns a value based on whether the first string is lexicographically less than, greater than, or the same as the second string, ignoring case. For example, the following program compares two strings and prints the results if the strings are equal:

```
#include <stdio.h>
#include <string.h>

main()
{
 int result;
 char *str3="compUter";
 char *str4="CoMputeR";

if (stricmp(str3,str4)==0)
 printf("Strings %s and %s are equal (case-insensitive)\n",
   str3,str4);
}
```

Returns

The `stricmp` function returns an integer greater than, equal to, or less than `0`, depending on whether the string pointed to by *s1* is lexicographically greater than, equal to, or less than the string pointed to by *s2*, ignoring case in both strings. This function does not return any special value to indicate an error.

Related Topics

`memicmp`
`strcmp` (*C: A Reference Manual*)
`strnicmp`

strlwr, strupr

Convert string to lower or upper case.

```
char *strlwr (char *s);
char *strupr (char *s);
```

s points to the string to be converted.

Header File `string.h`

Discussion

Use the `strlwr` function to convert any uppercase alphabetic characters in the string, pointed to by *s*, to lowercase.

Use `strupr` to convert any lowercase alphabetic characters in the string, pointed to by *s*, to uppercase.

These functions modify strings without moving them, so their input and return values are the same. These functions resemble the ANSI `tolower` and `toupper` functions, but apply to an entire string rather than a single character.

Returns

The `strlwr` and `strupr` functions return a pointer to the modified string. This function does not return any special value to indicate an error.

Related Topics

`tolower`	(*C: A Reference Manual*)
`toupper`	(*C: A Reference Manual*)

strnicmp

Compare strings, ignore case.

```
int strnicmp (const char *s1, const char *s2, size_t n);
```

s1, *s2*	point to the strings to be compared.
n	is the maximum number of characters in the strings to be compared.
Header File	`string.h`

Discussion

Use this function to compare two strings lexicographically, ignoring distinctions between lowercase and uppercase. The `strnicmp` function is a case-insensitive version of the ANSI `strncmp` function. As such, this function compares up to *n* characters of the first null-terminated string to the second and returns a value based on whether the first string is lexicographically less than, greater than, or the same as the second string (ignoring case).

For example, the following program compares two strings and prints the results if the strings are equal:

```
#include <string.h>
#include <stdio.h>

main()
{
 char *str1="hello world";
 char *str2="HELLO";
 char *str3="compUting";
 char *str4="CoMputeR";

 if (strnicmp(str1,str2,5)==0)
 {  printf("The first 5 characters of the strings %s",str1);
    printf(" and %s are equal (case-
    insensitive).\n",str2);
 }
 if (strnicmp(str3,str4,6)==0)
 {  printf("The first 6 characters of the strings %s",str3);
    printf(" and %s are equal (case-
    insensitive).\n",str4);
 }
}
```

Returns

The `strnicmp` function returns an integer less than, greater than or equal to zero depending on whether the first *n* characters of the string pointed to by *s1* are less than, greater than or equal to the first *n* characters of the string pointed to by *s2*. This function does not return any special value to indicate an error.

Related Topics

memicmp
stricmp
strncmp (*C: A Reference Manual*)

strnset

Set characters in string.

```
char *strnset (char *s, int c, size_t n);
```

s	points to the string to be set.
c	is the character-coded integer value to be assigned to characters in the string.
n	is the number of characters to be set.
Header File	`string.h`

Discussion

Use this function to set *n* number of characters of the string *s* to the value *c*.

Returns

The `strnset` function returns a pointer to the string. This function does not return any special value to indicate an error.

Related Topic

strset

strrev

Reverse characters in string.

```
char *strrev (char *s);
```

s	points to the string to be reversed.
Header File	`string.h`

Discussion

Use this function to reverse the order of characters in the string pointed to by `s`, leaving the terminating null character at the end.

Returns

The `strrev` function returns the pointer to the modified string. This function does not return any special value to indicate an error.

strset

Set characters in string.

```
char *strset (char *s, int c);
```

s	points to the string to be set.
c	is the character-coded integer value to be assigned to the characters in the string.
Header File	`string.h`

Discussion

Use this function to set all the characters in the string pointed to by *s*, except the required terminating null character, to the value *c*.

Returns

The `strset` function returns a pointer to the string. This function does not return any special value to indicate an error.

Related Topic

`strnset`

tzset

Set time zone variables.
SVID

```
void tzset (void);
```

Header File `time.h`

Discussion

Use this function to set the values of the following macros:

`daylight` provides the daylight savings time flag. The flag value is `0` if daylight savings time is in effect and nonzero otherwise. The default value is `1`. The `daylight` value has the type `int`.

`timezone`	provides the difference, in seconds, between Greenwich Mean Time (GMT) and local time. For example, the `timezone` value for Eastern Standard Time (EST) is `18000`. The `timezone` value has the type `long`.
`tzname`	provides a pair of strings identifying the time zone. The data type of each `tzname` value is declared as follows: `extern char *tzname[2]` The default value of `tzname[0]` is `PST`, indicating Pacific Standard Time, and of `tzname[1]` is `DST`, indicating daylight savings time.

The `tzset` function uses the `TZ` environment variable, specifying the relevant system time zone, to set the values of the `daylight`, `timezone`, and `tzname` global variables. The value of `TZ` must be in the form:

aaan[*bbb*]

aaa and *bbb*	are sequences of three arbitrary characters.
n	is the signed difference in hours from Greenwich Mean Time. A negative value indicates a location east of Greenwich, England.

The *bbb* string is optional. Including *bbb* indicates that daylight savings time is currently in effect. The default value for `TZ` is `PST8`.

For example, when `daylight` is `1`, `TZ` is `EST5EDT` for New York, `CST6CDT` for Illinois, `MST7MDT` for Colorado, and `PST8PDT` for Oregon.

Related Topics

`time` (Chapter 5)
`time.h` (Chapter 3)

ultoa, utoa

Convert unsigned long to string.
Convert unsigned integer to string.

`char *ultoa (unsigned long value, char *string, int radix);`

`char *utoa (unsigned int value, char *string, int radix);`

value	is the value to be converted.
string	is a pointer to the string.
radix	is the radix of *value*, in the range 2 through 36 decimal.
Header File	`stdlib.h`

Discussion

Use `ultoa` to convert the `unsigned long` value *value* to the equivalent null-terminated character string and store the result in *string*. Use `utoa` to convert the `unsigned int` value *value* to the equivalent null-terminated character string and store the result in *string*. Specify the radix of conversion with the *radix* argument, which must be in the range `2` through `36` decimal.

For example, the following program converts a value to a string and prints it:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  unsigned int val;
  char *buffer;
  char * buf;
```

```
  buffer=malloc(10);
  val=0x5689;
  buf=utoa(val,buffer,4);    /* buffer is "11122021" */
  fputs(buffer,stdout);
  free(buffer);
}
```

The *string* buffer must be large enough to hold the ASCII representation of the largest integer possible in your execution environment.

For portability, use `sprintf` with the `%lo`, `%ld`, or `%lx` conversion specifiers, if *radix* is 8, 10, or 16, respectively, instead of calling `ultoa`. Use `sprintf` with the `%o`, `%d`, or `%x` conversion specifiers, if *radix* is 8, 10, or 16, respectively, instead of calling `utoa`.

Returns

The `ultoa` and `utoa` functions return pointers to the converted strings. These functions do not return any special value to indicate an error.

Related Topics

`ltoa`, `ltos`
`ltoh`
`sprintf` (*C: A Reference Manual*)

Customizing the Libraries 5

The libraries support reentrancy under environments using single-thread applications for supported monitors and evaluation boards. Environments other than the evaluation boards directly supported by the MON960 retargetable monitor require retargeting of the low-level, environment-dependent libraries. Additionally, use of monitors not supported by the supplied board-specific libraries forces retargeting of the low-level, board-dependent libraries.

This chapter describes several types of reentrancy and explains how to rewrite low-level library functions and system calls for applications that use an unsupported board, monitor, or type of reentrancy.

Creating custom monitor libraries requires attention to the C run-time library reentrancy material presented in the Making the Libraries Reentrant section, which includes the following:

- how concurrent tasks and functions can share data without conflict
- how rewritten low-level functions must operate

Retargeting an application to run on other than a MON960-supported evaluation board requires attention to the retargeting information in Retargeting the Libraries section, which includes the following:

- how the library functions use system calls
- how rewritten system calls must operate

Making the Libraries Reentrant

This section

- defines reentrancy and associated terms
- describes the problems of persistent data
- describes the actions a newly written reentrant function must perform
- lists stubs to act as guides for the writing of new low-level routines.

This section assumes familiarity with the environment in which a new application will run and some familiarity with the issues of reentrancy.

Reentrancy Defined

This section contains a list of terms and definitions used in the discussion of reentrancy, a general definition of reentrancy, and a description of persistent data. The information in this section supports the writing of reentrant functions.

Terms

The remainder of this chapter uses the following terms:

context data	data that multiple threads can share that are directly referenced by functions.
multi-tasking execution environments	allow more than one task or process (referred to as a thread) to be active concurrently.
parallel reentrancy	two or more processes can execute a function simultaneously.
persistent data	consists of data structures and other variables that the libraries maintain outside of any function, to preserve data between function calls or to communicate data between functions. Persistent data can change during execution. The program allocates and

	initializes all persistent data structures as needed during startup and does not depend on a loader to store initial values.
recursive reentrancy	a process can suspend one instance of the function, start and execute another instance to completion, and reactivate the suspended instance.
thread	an independent execution of code that has its own instruction pointer and stack. For example, in a simple embedded control application, an interrupt handler constitutes a separate thread of execution.
thread data	data unique to the thread. The data cannot be shared.
time-slice reentrancy	execution can alternate or rotate between two or more processes executing the function. One process is active and the others are suspended at any given time.

Types of Reentrancy

A reentrant function can be active in two or more instantiations at once. In all cases of reentrancy, any given instance of the function must be able to operate on memory locations and processor registers without destroying the memory and register values used by any suspended or concurrent instantiation.

For example, in a multi-tasking environment, a reentrant function can be called from two or more concurrent threads without causing conflicting updates to the data structures used by the function.

The three types of multiple instantiation follow:

Parallel Two or more processes can execute a function simultaneously. Multi-tasking execution environments allow more than one task or process (referred to as a thread) to be active concurrently.

Time-sliced Execution can alternate or rotate between two or more processes executing the function. One process is active and the others are suspended at any given time.

Recursive A process can suspend one instance of the function, start and execute another instance to completion, and reactivate the suspended instance.

Persistent Data

Of the types of data the libraries use, only persistent data presents a problem for reentrancy. Because persistent data exists outside the function, separate instantiations of a function must not destroy data needed by other instantiations. Persistent data occurs in the following two forms:

Thread data must be unique to the thread and cannot be shared. This category includes, for example, the `errno` variable, the random number seed, and buffers containing structure and string return values of specific C functions. Thread data can be modified as a side effect rather than as the primary intent of a function call.

Context data is the only data directly referenced by functions that multiple threads can share. You can directly reference other shared data through pointers passed to functions, but data referenced in that way is not protected.

The context of a thread is the data space that can be shared between concurrent threads, and context data is shared between two or more threads in a context. The two classes of shareable data are:

- The exit handler and open I/O stream lists.
- Currently open streams, including the standard streams `stdin`, `stdout`, `stderr`.

The libraries process open streams independently of the clean-up lists that `exit` processes. All threads in a single context can share streams or each thread can have its own streams.

Writing Reentrant Functions

This section contains criteria and procedures necessary for writing reentrant functions and low-level reentrancy support functions. This section contains:

- general requirements for reentrant functions
- prerequisites for ROM based reentrant functions
- a list of actions each new function must perform
- a detailed discussion of each action
- tables of low-level memory handling functions and existing library functions which do not support reentrancy.

General Reentrancy Requirements

Reentrancy is possible when references to persistent data are made under the following conditions:

- Data is not shared between processes.
- References are controlled by preventing other processes from updating the data in conflicting ways.

The portable functions in the libraries are reentrant and support reentrant use of their data if the execution environment provides reentrant supporting access functions. Since the access functions in the libraries do not support reentrant operation, you must replace these functions with access functions appropriate to your execution environment.

There are four categories of reentrancy:

Category 1: Reentrant

- These functions call no other functions that are not known to be reentrant.
- All variables are local, stored on the stack or in a register.
- Functions can read statically allocated constant data.
- Functions can read and write data pointed to by parameters that were passed to the function. In such cases it is the caller's place to assure that the data is correctly accessed/protected if the function is reentered.

Category 2: Reentrant Except for Setting errno

- These routines are reentrant except for their setting of the `errno` variable.

Category 3: Reentrant Except for Setting fpem_CA_AC

- These routines are reentrant if at interrupt, or thread context change, the current state of the `fpem_CA_AC` is saved and restored. Note that for K- and S-series processors, there is no `fpem_CA_AC`, and therefore these routines are all reentrant for these processors.

Category 4: Non-reentrant

- Uses statically allocated variables that are not accessed via thread data structure.

Category 5: Unspecified

- Uses statically allocated variables that are accessed via thread data structure.
- Any routine that does IO is unspecified.

Using these categories, the entry points of the standrd C and math libraries and the accelerated and alternate floating-point libraries are categorized in the following tables. Note that some functions are in two categories (e.g., `sscanf` is in both Category 1 and Category 2).

Table 5-1 Category 1: Reentrant Functions

		libc C Library		
_getch	ediv	itoa	qsort	strrchr
_Ldoprnt	feof	itoh	setlocale	strrev
_Lmodeparse	ferror	labs	strcat	strrpos
_putch	isalnum	ltoa	strchr	strrpos
_thread_init	isalpha	ltoh	strcmp	strset
_tolower	isascii	ltos	strcoll	strspn
_toupper	iscntrl	mblen	strcpy	strstr
abs	isdigit	mbstowcs	strftime	strupr
atoi	isdigit	mbtowc	stricmp	strxfrm
bcmp	isgraph	memchr	strlen	system
bcopy	islower	memcmp	strlwr	tolower
bcopy	isodigit	memcpy	strnicmp	toupper
bsearch	isprint	memicmp	strnset	ultoa
bzero	ispunct	memmove	strpos	utoa
clock	isspace	memset	strpos	wcstombs
div	isupper	mktime	strrchr	wctomb
		libm Math Library		
__clsdfsi	_Lclog2xf	_Lmatherr	_Lylog2xl	fabsl
__clssfsi	_Lclog2xl	_Lratan2	atan	fmod
__clstfsi	_Lclogep2x	_Lratan2f	atanf	fp_clriflag
_AFP_dp2a	_Lclogep2xf	_Lratan2l	atanl	frexp
_AFP_mZERO_S	_Lclogep2xl	_Ls_do_mul	copysign	modf
_AFP_tp2a	_Ld_do_mul	_Lsatan2	copysignf	sinl
_Lclass	_Lexp2m1	_Lsatan2f	copysignl	square
_Lclassf	_Lexp2m1f	_Lsatan2l	cosl	tanl
_Lclassl	_Lexp2m1l	_Lylog2x	fabs	
_Lclog2x	_Lhypot_util	_Lylog2xf	fabsf	

continued ☛

Table 5-1 Category 1: Reentrant Functions (continued)

libh Accelerated Floating-point Library	
__fixdfsi	AFP_Fault_Invalid_Operation_S
__fixsfsi	AFP_Fault_Invalid_Operation_T
__fixtfsi	AFP_Fault_Overflow_D
__fixunsdfsi	AFP_Fault_Overflow_S
__fixunssfsi	AFP_Fault_Overflow_T
__fixunstfsi	AFP_Fault_Reserved_Encoding_D
__floatsidf	AFP_Fault_Reserved_Encoding_S
__floatsitf	AFP_Fault_Reserved_Encoding_T
__floatunssidf	AFP_Fault_Underflow_D
__floatunssitf	AFP_Fault_Underflow_S
AFP_Fault_Inexact_D	AFP_Fault_Underflow_T
AFP_Fault_Inexact_S	AFP_Fault_Zero_Divide_D
AFP_Fault_Inexact_T	AFP_Fault_Zero_Divide_S
AFP_Fault_Invalid_Operation_D	AFP_Fault_Zero_Divide_T

libfp Alternate Floating-point Library				
__absdf2	__fixsfsi	__subsf3	dplog	fpatn
__abssf2	__fixunsdfsi	__truncdfsf2	dpsin	fpcos
__adddf3	__fixunssfsi	__truncdfsf2_g960	dpsqrt	fpexp
__addsf3	__floatsidf	ceilf	dptan	fpln
__cmpdf2	__floatsisf	dascbin	dpxtoi	fplog
__cmpsf2	__muldf3	dbinasc	eptodp	fpsin
__divdf3	__mulsf3	dpatn	faint	fpsqrt
__divsf3	__negdf2	dpcos	fascbin	fptan
__extendsfdf2	__negsf2	dpexp	fbinasc	fpxtoi
__fixdfsi	__subdf3	dpln	floorf	

Table 5-2 Category 2: Reentrant Except for Setting errno

		libc C Library		
atol	sprintf	strerror	strtoul	
ldiv	sscanf	strtol	vsprintf	
		libm Math Library		
_AFP_INF_D	_Lqerrorf	atan2l	log	sinh
_AFP_INF_S	_Lqexpm1	atof	log10	sqrt
_AFP_int_pow	_Lstrtoe	cos	log10f	sqrtf
_AFP_int_powf	_Lstrtof	cosf	log10l	sqrtl
_AFP_NaN_D	acos	cosh	logf	strtod
_AFP_NaN_S	acosf	exp	logl	tan
_AFP_QNaN_D	asin	expf	pow	tanf
_AFP_QNaN_S	asinf	expm1	powf	tanh
_Lfltscan	atan2	hypot	sin	
_Lqerror	atan2f	ldexp	sinf	

Table 5-3 Category 3: Reentrant Except for Setting fpem_CA_AC

		libc C Library		
sprintf	sscanf	vsprintf		
		libm Math Library		
__Lnan1	_Lisnan	difftime	fp_remf	log10
__Lnan1f	_Lisnanf	exp	fp_reml	log10f
__Lnan1l	_Lisnanl	expf	fp_rmd	log10l
_getac	_Lqerror	expm1	fp_rmdf	log1p
_IEEE_sqrt	_Lqexpm1	expm1f	fp_rmdl	log1pf
_IEEE_sqrtf	_Lquickexit	floor	fp_round	log1pl
_Lfaultexit	_Lquickexitf	floorf	fp_roundf	logf
_Lfaultexitf	_Lquickexitl	floorl	fp_roundl	logl
_Lfaultexitl	_setac	fp_clrflags	fp_scale	pow
_Lflt_interface	acos	fp_getenv	fp_scalef	powf
_Lfltprnt	acosf	fp_getflags	fp_scalel	sinh
_Lfltscan	asin	fp_getmasks	fp_setenv	sqrt
_Lfpd_exit	asinf	fp_getround	fp_setflags	sqrtf
_Lfpe_exit	ceil	fp_logb	fp_setmasks	tanh
_Lfpi_exit	ceilf	fp_logbf	fp_setround	
_Lfpi_quickexit	ceill	fp_logbl	hypot	
_Lfps_exit	cosh	fp_rem	log	

continued ☛

Table 5-3 Category 3: Reentrant Except for Setting fpem_CA_AC (continued)

libh Floating-point Library				
___extenddftf2	__floatunssisf	__rmddf3	__subsf3	ceill
___extendsfdf2	__floordf2	__rmdsf3	__subtf3	floor
___extendsftf2	__floorsf2	__rmdtf3	__truncdfsf2	floorf
__adddf3	__floortf2	__rounddf2	__truncdfsf2_g960	floorl
__addsf3	__logbdf2	__rounddfsi	__trunctfdf2	fp_clrflags
__addtf3	__logbsf2	__roundsf2	__trunctfsf2	fp_clriflag
__ceildf2	__logbtf2	__roundsfsi	AFP_NaN_D	fp_getenv
__ceilsf2	__muldf3	__roundtf2	AFP_NaN_S	fp_getflags
__ceiltf2	__mulsf3	__roundtfsi	AFP_NaN_T	fp_getmasks
__cmpdf2	__multf3	__roundunsdfsi	AFP_RRC_D	fp_getround
__cmpsf2	__remdf3	__roundunssfsi	AFP_RRC_D_2	fp_setenv
__cmptf2	__remsf3	__roundunstfsi	AFP_RRC_S	fp_setflags
__divdf3	__remtf3	__scaledfsidf	AFP_RRC_S_2	fp_setmasks
__divsf3	__rintdf2	__scalesfsisf	AFP_RRC_T	fp_setround
__divtf3	__rintsf2	__scaletfsitf	ceil	
__floatsisf	__rinttf2	__subdf3	ceilf	

Table 5-4 Category 4: Non-reentrant

libc C Library			
free	localeconv	raise	tmpfile
getenv	localtim	realloc	tmpnam
getopt	malloc	signal	tzset
libm Math Library			
ecvt	fcvt	gcvt	

Table 5-5 Category 5: Unspecified

		libc C Library		
_assert	exit	fprintf	getchar	putw
_exit_init	fclose	fputc	gets	rand
_filbuf	fcloseall	fputchar	getw	remove
_flsbuf	fdopen	fputs	gmtime	rewind
_HL_init	fflush	fread	init_c	rmtmp
_Ldoscan	fgetc	freopen	lfind	scanf
_stdio_init	fgetchar	fscanf	lsearch	setbuf
abort	fgetpos	fseek	perror	setvbuf
asctime	fgets	fsetpos	printf	strtok
atexit	fileno	ftell	putc	ungetc
clearerr	flushall	fwrite	putchar	vfprintf
ctime	fopen	getc	puts	vprintf

Note that all routines under the C++ Iostream library are also considered unspecified.

ROM Reentrancy Requirements

If your application executes in read-only memory (ROM), any libraries you use must be written and compiled so that they meet the following constraints:

- You can place only constants in the code segment in ROM.
- You must place data that can change during execution in the data segment in random-access memory (RAM).
- You must place the instructions that initialize RAM data in the code segment in ROM.
- Each library you use meets all of the constraints for programming into ROM.

Contents of Reentrant Functions

To avoid data conflicts, the following three criteria must be true for newly written functions:

- Startup routines must initialize a context.
- The new function must create and maintain its own data pointers.
- The new function must call semaphores to protect itself from the influence of other instantiations.

Initializing a New Context

Startup code must initialize both thread data and context data for reentrant and ROM applications. To start a new context, your startup code must call the thread-initialization functions in the following order:

1. `_thread_init` initializes non-shared data.
2. `_exit_init` initializes memory for the exit handler.
3. `_stdio_init` initializes the standard I/O streams.

Both the startup code for the context and the initialization code for each thread must call `_thread_init`. A new thread starting within an existing context initializes only the data that it does not share. A new thread can call `_exit_init`, `_stdio_init`, or both, depending on the data that it shares, as follows:

- If a single call to `exit` is to terminate all threads within a context, then:
  - The startup code for the context must call `_exit_init` exactly once.
  - Subsequent threads in the context must not call `exit_init`.
- If `exit` is to terminate only the thread that calls it, then each thread in the context must call `_exit_init`.
- When two or more threads of a context share standard I/O streams (`stdin`, `stdout`, and `stderr`), the startup code for the context must call `_stdio_init` exactly once to initialize the context for those threads. Any thread that has its own standard streams must call `_stdio_init`.

Each of these initialization functions calls a corresponding function to allocate memory for the data. Since these functions, declared in the header file `reent.h`, depend on the execution environment, you must implement versions appropriate to your execution environment. The file `_create.c` contains sample source code for these functions in a single-thread (not reentrant) implementation. The memory allocation functions are:

`_exit_create`	allocates memory for the exit handler, either local to the thread or global within the context.
`_stdio_create`	allocates I/O buffers for the standard I/O streams, either local to the thread or global within the context.
`_thread_create`	allocates data space for the thread. This block of memory is associated only with the calling thread.

Each of these initialization functions operates as a special-purpose `malloc` function: the function takes an argument that specifies the amount of memory requested and returns a pointer to a block of memory at least that big. The calling thread then owns that block of memory.

To finish initializing the standard streams, `_stdio_init` also calls the function `_stdio_stdopen`. When called with an argument of `0`, `1`, or `2`, `_stdio_stdopen` returns the file number associated with `stdin`, `stdout`, or `stderr`, respectively.

NOTE. *Make sure replacement startup code calls the initialization functions listed in Table 5-2. The table lists the functions, the libraries in which each is located, and the action of the function. These low-level functions make no additional calls which require attention. For a list of additional functions and the calls each function makes, refer to Appendix A.*

Table 5-6 Memory Handling Functions for Reentrancy

Usage	Name	Operation
initialization (These functions are in the high-level libraries.)	_exit_init	Initializes the exit handler for a new thread in a context.
	_stdio_init	Initializes the standard I/O streams for a new thread in a context.
	_thread_init	Initializes non-shared data for a new thread in a context.
memory allocation (These functions are in the MON960 debug monitor library.)	_exit_create	Allocates memory for the exit handler.
	_stdio_create	Allocates standard stream buffers associated with a given thread.
	_thread_create	Allocate for a given thread.
memory access (These functions are in the MON960 debug monitor library.)	_exit_ptr	Returns a pointer to exit lists.
	_stdio_ptr	Returns pointers to the standard streams.
	_thread_ptr	Returns a pointer to the thread data space.
	_tzset_ptr	Returns a pointer to the _tzset structure containing time zone information.
synchronization (These functions are in the MON960 debug monitor library.)	_semaphore_delete	Frees resources associated with a semaphore.
	_semaphore_init	Initializes a semaphore for a multi-tasking context.
	_semaphore_signal	Releases a memory location.
	_semaphore_wait	Queues requests for access to a memory location.

Creating Pointers to Data

All library functions that access thread or context data use one of the following access functions to obtain a pointer to the data:

`_errno_ptr`	returns a pointer to the `errno` flag.
`_exit_ptr`	returns a pointer to the exit lists.
`_stdio_ptr`	returns a pointer to the standard streams.
`_thread_ptr`	returns a pointer to the block of memory unique to the calling thread.

To return the same pointers as `_exit_create`, `_stdio_create`, and `_thread_create` for the current thread, the access functions you write must use the information used by the execution environment to manage the threads of execution. The file `_create.c` contains sample source code for these functions in a single-thread (not reentrant) implementation.

The `errno` macro contains a value indicating the cause of the most recent error that has occurred in execution. The address of `errno` is the value returned from the `_errno_ptr` and `_thread_create` functions. Any function that can set `errno` must be able to write to that address.

Calling Semaphore Functions

To prevent different threads from performing conflicting updates, functions that access context data must call the following semaphore functions:

`_semaphore_delete`	frees resources associated with a semaphore.
`_semaphore_init`	initializes a semaphore for a context.
`_semaphore_wait`	queues requests for access to a memory location.
`_semaphore_signal`	releases a memory location.

The _semaphore_init function initializes a semaphore. Library functions later call _semaphore_wait before updating the associated data. All but the first call to _semaphore_wait with a given address must be queued for access to that address until the function using the data releases the address by calling _semaphore_signal. Depending on the environment, the implementation of _semaphore_init need not be as comprehensive as the complete interface between threads of a context. For example, if threads can share I/O streams but exit terminates only the thread that calls it, then _semaphore_wait needs to be used only to synchronize access to a stream, not to coordinate the exit lists.

If threads of a context share exit handlers and share open-stream lists but do not share streams, you can implement the semaphore-queueing functions as follows:

- If the address passed to _semaphore_wait is within the region allocated by _exit_create, then either the exit-handler list or the open-file list is currently being manipulated.
- If the address passed to _semaphore_wait is not within the region allocated by _exit_create, then a stream is currently being accessed.

Alternatively, you can implement _semaphore_wait simply so that it disables interrupts and _semaphore_signal so that it re-enables them. However, this simpler implementation cannot work in an environment where I/O is interrupt-driven.

NOTE. *The macro implementations of* `getc`, `getchar`, `putc` *and* `putchar` *do not invoke semaphore operations.*

The library allocates void pointers associated with each I/O stream, with the list of open streams, and with the list of exit handlers. Although the library functions never use these pointers, the addresses of these pointers are used as arguments to semaphore functions. You can specify what a semaphore function stores in any pointer. For example, as an additional context to support semaphores, your `_semaphore_init` can allocate a block of memory and reference the memory through a pointer.

The file `_semaph.c` contains sample source code for these functions in a single-thread (not reentrant) implementation.

To provide reentrancy, you must replace the stub semaphore functions in the libraries with functions appropriate to your execution environment.

The stub semaphore functions are:

```
_semaphore_delete
_semaphore_init
_semaphore_signal
_semaphore_wait
```

Primitive Function Descriptions

The low-level functions in the libraries do not depend on a particular operating system and are designed for single-thread (not reentrant) execution. If your execution environment supports memory sharing between concurrent processes, then you must replace the library of single-thread functions with a library that supports reentrant execution. Source file templates for some of the low-level functions are supplied with the libraries. The low-level templates are in these files in src/lib/libll/common:

```
_arg_ini.c          isatty.c
c_init.c            _map_len.c
_create.c           _semaph.c
c_term.c            _stdopen.c
_def_sig.c          _tzset.c
getend.c
```

This section lists function descriptions to help you implement replacements for library functions. The header files listed with the function descriptions provide the macros, function prototypes, and other symbols used by the functions. Appendix A shows which high-level libraries call these primitive functions.

NOTE. *A few low-level functions only call additional low-level functions. Because they only call other functions, they need not be rewritten. A note appears in the discussion section of the functions which do not need to be rewritten.*

_arg_init

Sets up the argv and argc arguments for the main function.

```
struct { int argc; char ** argv }  _arg_init(void);
```

Header File None required

Discussion

This function sets up the `argv` and `argc` arguments for the `main` function.

Returns

The `_arg_init` function returns the appropriate value for the first parameter to `main(argc)` in `g0`, and the appropriate value for the second parameter to `main(argv)` in `g1`.

Related Topic

`_HL_init`

_errno_ptr

Get a pointer to the errno variable.

```
struct _stdio  *_errno_ptr (void);
```

Header File `reent.h`

Discussion

This function provides a pointer to `errno` variable for the current thread.

Returns

The address of the `errno` variable for the current thread.

Related Topics

None.

_exit_create

Allocate space for exit list.

```
struct _exit  *_exit_create (unsigned nbyte);
```

`nbyte` is the amount of memory in bytes requested.

Header File `reent.h`

Discussion

This function allocates *nbyte* bytes of memory, associates the allocated space with the thread of execution from which it was called, and returns a pointer to the allocated space. Any subsequent call to the function `_exit_ptr` from the same thread must return the same pointer.

If `exit` terminates all threads in a context, the startup code must call `_exit_create` exactly once and `_exit_create` need not associate the memory it allocates with a particular thread. If `exit` terminates only the calling thread, `_exit_create` must be called for each thread as it is established.

NOTE. *The library functions require the* `_exit` *structure as declared in the header file* `reent.h`.

Returns

The `_exit_create` function returns a pointer to an area of memory at least *nbyte* bytes long.

Related Topics

```
exit, _exit
_exit_init
_exit_ptr
```

_exit_init

Initialize exit handler.

```
int _exit_init (void):
```

Header File `reent.h`

Discussion

This function calls `_exit_create` to allocate space for the `_exit` structure and initializes `_exit` as follows:

- sets the open-file list pointer to `null`
- sets the exit-handler count to `0`

You need not rewrite this high-level function.

NOTE. *The library functions require the* `_exit` *structure as declared in the header file* `reent.h`.

Returns

The `_exit_init` function returns no value.

Related Topics

```
exit, _exit
_exit_create
_exit_ptr
```

_exit_ptr

Get a pointer to the exit handler list.

```
struct _exit  *_exit_ptr (void);
```

Header File `reent.h`

Discussion

This function returns the same pointer as `_exit_create` if called from the same thread. This pointer points to the memory space allocated by `_exit_create`. If `exit` terminates all threads in a context, `_exit_ptr` need not return a unique pointer for each thread.

NOTE. *The library functions require the* `_exit` *structure as declared in the header file* `reent.h`.

Returns

The `_exit_ptr` function must return the same pointer as did `_exit_create` when called by this thread.

Related Topics

`exit`, `_exit`
`_exit_create`
`_exit_init`

_HL_init

Perform high-level library initializations.

```
void _HL_init (void);
```

Header File None required

Discussion

This function, included in the architecture-specific `libcxx.a` high-level libraries, performs all necessary high-level library initializations. These initializations ensure correct operation of all library functions, including any I/O functions such as `printf`. The `_HL_init` function calls the `_exit_init`, `stdio_init`, and `_thread_init` functions.

You need not rewrite this high-level function.

Returns

The `_HL_init` function returns no value.

Related Topics

```
_arg_init
_exit_init
_LL_init
_stdio_init
_thread_init
```

_LL_init

Perform low-level library initializations.

```
void _LL_init (void);
```

Header File None required

Discussion

This function, included in the board-specific low-level libraries, performs all necessary chip and board initialization functions. For example, in addition to initializing the i960 data structures, the startup function must set `mem_end` to point to the end of available memory used by `sbrk`.

Returns

The `_LL_init` function returns no value.

Related Topics

`brk, sbrk`
`_HL_init`

_semaphore_delete

Delete semaphores.

```
void  _semaphore_delete (void **);
```

Header File `reent.h`

Discussion

This function frees any resources attached to the semaphore associated with the pointer argument.

Returns

The `_semaphore_delete` function returns no value.

Related Topics

```
_semaphore_init
_semaphore_signal
_semaphore_wait
```

_semaphore_init

Initialize semaphore.

```
void  _semaphore_init (void **);
```

Header File `reent.h`

Discussion

This function creates and initializes a unique semaphore associated with the pointer argument. The high-level library calls `_semaphore_init` before using any other semaphore operation. Use semaphore operations to control updates to context data.

Returns

The `_semaphore_init` function returns no value.

Related Topics

```
_HL_init
_semaphore_delete
_semaphore_signal
_semaphore_wait
```

_semaphore_signal

Release a semaphore.

```
void  _semaphore_signal (void **);
```

Header File `reent.h`

Discussion

This function releases the semaphore associated with the pointer argument as flow of execution leaves a critical section of the code or as an operation finishes using a critical memory location. Releasing the semaphore allows a waiting thread to enter the critical section of the code or access the critical memory location.

NOTE. *The macro implementations of* `getc`, `getchar`, `putc`, *and* `putchar` *do not use semaphore functions.*

Returns

The `_semaphore_signal` function returns no value.

Related Topics

```
_semaphore_delete
_semaphore_init
_semaphore_wait
```

_semaphore_wait

Enter a critical region.

```
void  _semaphore_wait (void **);
```

Header File `reent.h`

Discussion

This function acquires the semaphore associated with the pointer argument if the semaphore is free. Otherwise, `_semaphore_wait` suspends the calling thread until `_semaphore_signal` releases the semaphore. If more than one thread can call `_semaphore_wait` with the same pointer before that semaphore becomes free, you must implement some form of thread-queueing mechanism.

NOTE. *The macro implementations of* `getc`, `getchar`, `putc`, *and* `putchar` *do not use semaphore functions.*

Returns

The `_semaphore_wait` function returns no value.

Related Topics

`_semaphore_delete`
`_semaphore_init`
`_semaphore_signal`

_stdio_create

Allocate space for stream data.

```
struct _stdio  *_stdio_create (unsigned nbyte);
```

`nbyte` is the amount of memory in bytes to be allocated.

Header File `reent.h`

Discussion

This function allocates `nbyte` bytes of memory, associates the allocated space with the calling thread of execution, and returns a pointer to the space. A subsequent call to the function `_stdio_ptr` from the same thread must return the same pointer. If standard streams are shared between threads, the startup code must call `_stdio_create` exactly once and `_stdio_create` need not associate the memory it allocates with a particular thread.

NOTE. *The library functions require the* `_stdio` *structure as declared in the header file* `reent.h`.

This function is called by `_stdio_init`. Note also that this function can also perform other thread or context initialization required by the target environment.

Returns

The `_stdio_create` function returns a pointer to an area of memory at least `nbyte` bytes long.

Related Topics

```
_stdio_init
_stdio_ptr
_stdio_stdopen
```

_stdio_init

Initializes standard streams.

```
int _stdio_init (void)
```

Header File `reent.h`

Discussion

This function initializes the open-stream list with the following standard streams:

`stdin`	is the standard input stream.
`stdout`	is the standard output stream.
`stderr`	is the standard error stream.

You need not rewrite this high-level function.

Returns

The `_stdio_init` function returns no value.

Related Topics

```
_HL_init
_stdio_create
_stdio_ptr
_stdio_stdopen
```

_stdio_ptr

Get a set of pointers to the standard streams.

```
struct _stdio  *_stdio_ptr (void);
```

Header File `reent.h`

Discussion

This function provides a pointer to the data structure representing the standard streams for the calling thread. If two or more threads share standard streams, `_stdio_ptr` need not return a unique pointer for each thread.

NOTE. *The library functions require the* `_stdio` *structure as declared in the header file* `reent.h`.

Returns

The `_stdio_ptr` function must return the same pointer as `_stdio_create` when called by this thread.

Related Topics

`_stdio_create`
`_stdio_init`
`_stdio_stdopen`

_stdio_stdopen

Open a standard stream.

```
int  _stdio_stdopen (int str);
```

str	indicates which stream to open.
Header File	`reent.h`

Discussion

This function opens the standard stream and returns the associated file number. The argument *str* selects the stream to be opened, as follows:

`0`	selects `stdin`.
`1`	selects `stdout`.
`2`	selects `stderr`.

Returns

The `_stdio_stdopen` returns the file number for the selected standard stream.

Related Topics

`_stdio_create`
`_stdio_init`
`_stdio_ptr`

_thread_create

Allocate data space for a thread.

```
struct _thread  *_thread_create (unsigned nbyte);
```

Header File reent.h

Discussion

This function allocates *nbyte* bytes of memory, uniquely associates the allocated space with the current thread of execution, and returns a pointer to the allocated space. A subsequent call to the function _thread_ptr from the same thread must return the same pointer.

This function is called by _thread_init.

NOTE. *The library functions require the* _thread *structure as declared in the header files.*

Returns

The _thread_create function returns a pointer to an area of memory uniquely associated with the calling thread of at least *nbyte* bytes long.

Related Topics

_thread_init
_thread_ptr

_thread_init

Initialize thread data space.

```
int _thread_init (void);
```

Header File `reent.h`

Discussion

This function calls `_thread_create` to allocate space for the `_thread` structure and initializes `_thread` as follows:

- sets `errno` to `0`
- sets the random number seed to `1`

NOTE. *The library functions require the* `_thread` *structure as declared in the header file* `reent.h`.

You need not rewrite this high-level function.

Returns

The `_thread_init` function returns no value.

Related Topics

`_HL_init`
`_thread_create`
`_thread_ptr`

_thread_ptr

Get a pointer to thread data space.

```
struct _thread  *_thread_ptr (void);
```

Header File `reent.h`

Discussion

This function returns a pointer to the data structure for the calling thread.

NOTE. *The library functions require the* `_thread` *structure as declared in the header file* `reent.h`.

Returns

The `_thread_ptr` function must return the same pointer as `_thread_create` when called by this thread.

Related Topics

`_thread_create`
`_thread_init`

_tzset_ptr

Get time zone data.

```
struct _tzset  *_tzset_ptr (void);
```

Header File reent.h, time.h

Discussion

The structure _tzset is declared as follows:

```
struct _tzset
            {
             char *_tzname[2];
             long _timezone;
             int  _daylight;
            }
```

The timezone, daylight, and tzname macros and the localtime, strftime, ctime, and mktime functions call _tzset_ptr to obtain information about the effective time zone. The _tzset_ptr function uses the structure _tzset that contains members corresponding to timezone, daylight, and tzname.

If the effective timezone is not available in your execution environment, you can implement _tzset_ptr with a function that returns a NULL pointer.

Returns

The _tzset_ptr function returns a pointer to the _tzset structure containing time zone information. If the time zone information is not available in the execution environment, _tzset_ptr returns the NULL pointer value.

Related Topics

`time`
`time.h` (Chapter 3)

Retargeting the Libraries

To rewrite the library functions for a new execution environment, follow these steps:

1. Determine what environment-dependent library functions your application uses, both directly by calls in your source text and indirectly by calls from other library functions. Some of the environment-independent library functions depend on startup code to initialize data structures. The startup code in turn depends on operating system services and some environment-dependent functions. In restricted environments, some library functions are not useful or are not easy to implement. You need not implement functions that your application does not use.
2. Use the function descriptions in this manual and in *C: A Reference Manual*, to implement the new library functions.
3. Compile or assemble the new functions.
4. Create one or more new libraries with the new functions.
5. Link the new libraries to your application.

Function Interdependencies

See Table A-1 in the appendix for a list of functions that are directly or indirectly environment-dependent.

System Call Descriptions

This section describes the system calls for guidance in retargeting the libraries. These functions are not contained in the libraries of portable functions. The libraries provide the necessary functions for the Intel MON960 debug monitor-supported target environments. To use the libraries in a custom execution environment, you must provide system call functions appropriate for that environment.

close

Close a file.
POSIX 6.3.1

```
int close (int filedes);
```

filedes	is an open file descriptor.
Header File	`std.h`

Discussion

Use this function to close the file associated with the file descriptor *filedes*. The file descriptor is then available for reuse.

Returns

On successful completion, `close` returns `0`; otherwise, `close` returns `-1`.

Related Topics

`creat`
`fileno` (Chapter 4)
`open`

creat

Create a new file or rewrite an existing one.

```
int creat (char *path, int mode);
```

path	is a valid pathname for a file in the execution environment.
mode	is the permission setting which applies only to a newly created file.
Header File	`std.h`

Discussion

Use this function to create a new file, or to open and truncate an existing file, for writing. If `path` does not exist, `creat` creates a new file with the given mode settings then opens the file for writing; otherwise, `creat` truncates the file length to zero before opening the file for writing.

The permission setting, indicated by `mode`, only applies to a newly created file. `creat` sets the settings after closing the new file for the first time. You must specify one of the following access modes, as defined in the `fcntl.h` header file:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.
`O_RDWR`	Open for reading and writing.

Returns

Upon successful completion, `creat` sets the file pointer to the beginning of the file, and returns the new file number. Otherwise, `creat` returns `-1` and sets `errno` to `EACCES`, `EMFILE`, or `ENOENT`.

Related Topics

`close`	
`fileno`	(Chapter 4)
`open`	

_exit

Terminate a process.
POSIX 3.2.2

```
void _exit (int status);
```

`status`	is the value to be returned to the execution environment when the process terminates.
Header File	`std.h`

Discussion

Use this function to terminate the calling process and to close all files that are open in the calling process. The function `exit` calls `_exit` to terminate execution of a program without returning through all the currently active calling functions.

The `exit` function performs cleanup actions before the process exits. The `_exit` function circumvents any further cleanup.

The *`status`* value must be recognizable to the operating system or execution environment. By convention, a non-zero value indicates normal program termination.

Returns

The `_exit` function never returns to the program.

Related Topics

_exit_create
_exit_init
_exit_ptr

ioctl

Determines whether the I/O stream is a terminal device.

```
ioctl (int filnum, int com, int arg);
```

filnum	is a file number obtained from a `creat` or `open` system call.
com	is the function `ioctl` is to perform.
arg	is an argument specific to *com* if needed.
Header File	`ioctl.h`

Discussion

Use this function to determine whether or not an I/O stream is a terminal. The library only uses the first parameter, *filnum*. If you are rewriting your own low-level library, you can ignore the *com* and *arg* parameters. These two parameters exist for historical reasons and compatibility with UNIX.

Returns

Upon successful completion, `ioctl` returns a value greater than or equal to `0` if the I/O stream came from a terminal device. If not, `ioctl` returns a value less than `0`.

Related Topic

`isatty`

isatty

Identify a terminal device.
POSIX 4.7.2

```
int isatty (int filnum);
```

filnum	is a file number obtained from a `creat` or `open` system call.
Header File	`isatty.h`

Discussion

The `isatty` function identifies whether the file associated with *filnum* is a terminal device.

Returns

Upon successful completion, `isatty` returns a `1` if *filnum* is associated with a terminal device, and `0` otherwise. If *filnum* is an invalid file number, `isatty` returns `0` and sets `errno` to `EBADF`.

Related Topics

`creat`
`open`
`ioctl`

lseek

Move the read/write file pointer.
POSIX 6.5.3

```
long lseek (int filnum, long int offset, int whence);
```

filnum	is a file number obtained from a `creat` or `open` system call.
offset	is the number of bytes to increment the file pointer from the starting position.
whence	is the starting position of the file pointer.
Header File	`std.h`

Discussion

Use this function to change the file pointer associated with *filnum* using the following procedure:

1. Set the file pointer to the beginning of the file, to the end of the file, or leave the file pointer unchanged, according to *whence*, as follows:

   `0` set the file pointer to the beginning of the file

   `1` leave the file pointer at the current location

   `2` set the file pointer to the end of the file

2. Add the value of *offset* to the file pointer. The value of *offset* can be any positive, zero, or negative integer.

Returns

On successful completion, `lseek` returns the resulting offset in bytes from the beginning of the file; otherwise, `lseek` returns `-1` and sets `errno` to `EBADF`. An `lseek` operation on a non-disk file returns `-1`.

_map_length

Simulate file-to-stream mapping.

```
int  _map_length (int filnum, const void *buf,
                  size_t nbyte);
```

filnum	is the file number.
buf	is the input buffer for the stream.
nbyte	is the position of a character in the input buffer relative to the beginning of the buffer.
Header File	`std.h`

Discussion

Use this function to compensate for the mapping between characters in streams and files. The `ftell` function calls `_map_length` to compute one character's position in the stream buffer relative to its position in the file format supported by the execution environment. These positions can be different if, for instance, a carriage-return/newline pair is translated to a newline character (and vice-versa) on reading and writing ASCII characters. The `ftell` function obtains the approximate file position from `lseek`. Your implementation of `_map_length` must adjust this file position to agree with the number of bytes actually in the buffer, based on how input and output strings are processed in your application.

Since mapping is normally one-to-one for streams opened in binary mode, your implementation of `_map_length` can use *`filnum`* to obtain information about the file mode.

Returns

The `_map_length` function returns the number of characters needed to represent the *`nbytes`* of data in the buffer *`buf`*.

open

Open a file and set mode.
POSIX 5.3.1

```
int open (const char *path, int oflag [, mode_t mode]);
```

path	points to the pathname of the file to be opened.
oflag	indicates how the file is to be opened for reading and/or writing.
mode	is the access mode to be set for a new file. This argument is legal, and required, only when *oflag* includes `O_CREAT`, described below.
Header File	`std.h`, `fcntl.h`, `types.h`

Discussion

Use `open` to get a file descriptor which is associated with the file identified by *path*. The access modes and status flags of the open file descriptor are set according to *oflag*.

For *oflag*, you must specify one of the following access modes, defined in `fcntl.h`:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.
`O_RDWR`	Open for reading and writing.

See the discussion of the `fcntl.h` header file in Chapter 2 for definition of the POSIX file access mode macros.

In addition to the required access mode, you can also use one or more of the following file status flags in *oflag*:

O_APPEND	Perform all writes at the end of the file.
O_CREAT	Creates a new file, unless you specify O_EXCL and the file already exists.
O_EXCL	Used only with O_CREAT, returns an error value instead of opening any existing file.
O_TRUNC	Truncates any existing file named *path* to 0 bytes.

To use more than one status flag, you must add (+) or bitwise inclusive-OR (|) them together in the call to open.

Specify the third argument (mode_t *mode*) only if *oflag* includes O_CREAT. This argument is required with O_CREAT, but has no affect if the file identified by *path* already exists (see the discussion of O_EXCL). The *mode* argument sets the file permission bits for the file.

In addition to the POSIX file status flags, the following status flags are supported:

O_BINARY	Open in binary mode.
O_TEXT	Open in text mode.

These modes are mutually exclusive; do not OR them.

Returns

On successful completion, open returns the lowest numbered unused file descriptor. The file descriptor is used to reference the file in calls to the ioctl, isatty, close, lseek, read, and write functions. If an error occurs, open returns -1 and sets errno to EACCES, EEXIST, EMFILE, or ENOENT.

Related Topics

`close`	`lseek`
`creat`	`read`
`fcntl.h` (Chapter 3)	`stat.h` (Chapter 3)
`fstat`	`write`
`isatty`	

read

Read from a file.
POSIX 6.4.1

```
int read (int filnum, char *buf, unsigned int nbyte);
```

`filnum`	is a file number obtained from a `creat` or `open` system call.
`buf`	is the input buffer.
`nbyte`	is the number of bytes to be read.
Header File	`std.h`

Discussion

Use `read` to read *`nbyte`* bytes from the file associated with *`filnum`* into the buffer pointed to by *`buf`*.

Reading proceeds from the file position indicated by the file offset associated with *`filnum`*. The `read` function increments the file offset by the number of bytes written. If the file position, indicated before the read operation begins, is after the end of file, no bytes are read.

For example, if the text file representation of the operating environment does not exactly match the C stream representation (e.g., for newlines or tabs), the `read` function maps from the file representation to the stream representation for files opened in text mode.

Returns

Upon successful completion, `read` returns the number of bytes actually read and placed in the buffer. This number can be less than *`nbyte`* if the file is associated with a communication line or if the number of bytes left in the file is less than *`nbyte`* bytes. The `read` function returns zero on reaching end-of-file.

If the read operation does not complete successfully, `read` returns `-1` and sets `errno` to `EBADF`.

sbrk

Change data segment space allocation.

```
void *sbrk (unsigned incr);
```

`incr`	is the incremental change in number of bytes to the size of the data segment.
Header File	`std.h`

Discussion

Use `sbrk` to dynamically change the amount of space allocated for the data segment of the calling process. This function resets the break value of the process and allocates the requested space. The break value is the address of the first location beyond the end of the data segment. The size of the data segment increases as the break value increases.

The `sbrk` function adds *`incr`* bytes to the break value and changes the allocated space accordingly. Any newly allocated space is not initialized.

The `malloc` function calls `sbrk` when not enough memory is available in the heap to satisfy an allocation request. Memory allocated with `sbrk` cannot be freed or reallocated with `free` or `realloc`.

If the specified *incr* increases the size of the data segment above the system-imposed maximum, sbrk fails without changing the allocated space.

Returns

The sbrk function must return a quadword-aligned pointer. Upon successful completion, sbrk returns the address of the acquired memory area, that is, the old break pointer value. If the allocation request cannot be satisfied, either function returns -1.

Related Topic

malloc (*C: A Reference Manual*)

_sig_*

Provide signal handling.

```
void _sig_abrt_dfl(void);   /* abort */
void _sig_alloc_dfl(void);  /* allocation error */
void _sig_fpe_dfl(void);    /* floating-point exception */
void _sig_free_dfl(void);   /* bad free pointer */
void _sig_ill_dfl(void);    /* illegal instruction */
void _sig_int_dfl(void);    /* interrupt */
void _sig_read_dfl(void);   /* read error */
void _sig_segv_dfl(void);   /* segment violation */
void _sig_term_dfl(void);   /* software termination */
void _sig_write_dfl(void);  /* write error */

void _sig_null(void);    /* an unmasked signal occurred */
```

Header File signal.h

Discussion

The `raise` function uses each function, described above, as the default signal handler for the corresponding signal. Each signal handler takes the signal number of the raised signal as its argument.

Raising an ignored signal (i.e., one which is set to `SIG_IGN`) results in a call to `_sig_null` which takes no action.

Related Topics

```
raise
_HL_init
signal.h
```

stat

Obtain file status.
POSIX 5.6.2

```
int stat (char path, struct stat *buf);
```

path	is a pathname to a file. All directories in *path* must be searchable.
buf	is a pointer to a structure of type `stat`, into which information about the file is placed.
Header File	`stat.h`

Discussion

Use `stat` to get the status of the file identified by *`path`* and to store the information in the `stat` structure pointed to by `buf`. For example, the following program tests the status of a file:

```
#include <stdio.h>
#include <time.h>
#include <stat.h>

char filename[40];
main()
{
  char *date;
  int ret;
  struct stat buf;

  strcpy(filename,"testfile");

  if(ret=stat(filename,&buf)){
    fprintf(stderr,"stat failure error %d\n",ret);
    abort();
  }

  date=asctime(localtime(&buf.st_ctime));
  printf("\n %s",date);
  printf("\n %d mode",buf.st_mode);
  printf("\n %d size",buf.st_size);
}
```

The `stat` function stores the status information in the `stat` structure to which *`buf`* points. Useful members of the `stat` structure are:

`st_mode` is a bit mask in which:

- The `S_IFCHR` bit indicates that the file escriptor is associated with a character device.
- The `S_IFREG` bit indicates that it is associated with a normal file.
- The file permission bits indicate the mode in which the file is currently open.

`st_size`	indicates the size of a file. If the file descriptor refers to a character device, such as a printer or a console screen, this value is `1`.
`st_mtime`	contains the time and date of the last modification of the file. Use the time functions to interpret this value.
`st_atime`	contains the time and date of the last time the file was accessed. Use the time functions, declared in the `time.h` header file, to interpret this value.
`st_ctime`	contains the time and date of when the file was created. Use the time functions to interpret this value.

Chapter 3 lists status macros defined in the `stat.h` header file for use with the `stat` function.

Returns

On successful completion, `stat` returns `0`; otherwise, `stat` returns `-1` and sets `errno` to `EBADF`.

time

Get the system time.

```
time_t time (time_t *tloc);
```

`tloc`	points to a variable containing the system time.
Header File	`time.h`

Discussion

The `time` function returns the current time, measured in seconds since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970.

If the value of *`tloc`* is non-zero, the return value is stored in the location to which *`tloc`* points.

Returns

Upon successful completion, `time` returns the current system time.

Related Topics

`time.h`	(Chapter 3)
`tzset`	(Chapter 4)
`_tzset_ptr`	

unlink

Delete a filename.
POSIX 5.5.1

```
int unlink( char * filename);
```

`filename`	is the pathname of the file to be deleted.
Header File	`std.h`

Discussion

Use this function to delete the file specified by filename. This function performs the same task as the `remove` function, described in *C: A Reference Manual*.

Returns

On successful completion, `unlink` returns zero; otherwise, `unlink` returns a non-zero value.

write

Write to a file.
POSIX 6.4.2

```
int write (int filedes, const void *buf, unsigned nbyte);
```

`filedes`	is an open file descriptor.
`buf`	points to the buffer containing the bytes to be written to the file.
`nbyte`	is the number of bytes to be written to the file.
Header File	`std.h`

Discussion

Use `write` to write *`nbytes`* bytes from the buffer pointed to by *`buf`* to the file identified by the open file descriptor *`filedes`*.

Writing proceeds from the file position indicated by the file offset associated with *`filedes`*. The `write` function increments the file offset by the number of bytes written. If the result is greater than the length of the file, the file is extended.

The `O_APPEND` flag used with `creat` or `open` causes the offset to be set to the end of the file before writing begins.

If the text file representation of the operating environment does not exactly match the C stream representation, (e.g., for newlines or tabs), the `write` function maps from the stream representation to the file representation for files opened in the text mode.

Returns

On successful completion, `write` returns the number of bytes written to the file associated with *`filedes`*. This number is always less than or equal to *`nbyte`*. If `write` returns a number less than *`nbyte`*, an error occurred but some bytes were written. If `write` is unable to process any characters it returns `-1`, and sets `errno` to `EBADF` or `ENOSPC`.

Related Topics

`creat`
`open`
`stat.h` (Chapter 3)

Accelerated Floating-point Library 6

This chapter describes the accelerated floating-point library called "the AFP library" or "libh." (See Chapter 2 for a list of the actual library archive file names.)

Floating-point Library Definition

The accelerated floating-point library is a set of high speed assembly language subroutines that enable the i960 KA, SA, Cx, Jx, Hx, Rx or VH processors to perform floating-point operations. These processors do not support on-chip floating-point operations.

This library is used with the gcc960 and ic960 compilation system. When compiling for the processors without on-chip floating-point support, the compiler translates C language floating-point statements into assembly language instructions containing calls to libh subroutines.

The floating-point library is packaged as a collection of common object file format (ELF) subroutines. Several versions of the library are provided, as described in Chapter 2 of this manual.

To use a floating-point library, link your application with it. It should be the last library specified in the link sequence. Also, include the `afpfault.h` header file, which defines the interface to the stub routines provided for fault-handling. For more information on linking, see the *i960 Processor Software Utilities User's Guide*.

Assembly language programmers can place direct calls to the libh subroutines in their source text. The libh subroutines can also be called from C language source, but little is gained because the compiler optimizes C language floating-point code very efficiently. All examples in this manual show the subroutine names beginning with three underscore characters, as they appear in assembly language source. Use only two underscore characters if you call libh subroutines directly from C language source. The following examples highlight this difference:

`__addsf3`	for use in assembly language source.
`__addsf3`	for use in C language source.

To effectively use the floating-point library, you must understand the floating-point features of the KB processor, many of which are emulated in floating-point library subroutines.

Conventions

In this chapter, the following notation is used:

dst	the destination operand or return value of a subroutine.
src1	the first source operand or parameter of a subroutine.
src2	the second source operand or parameter of a subroutine.

The following definitions are also used throughout this manual:

integer	a two's complement 32-bit integer value.
unsigned integer	an unsigned 32-bit integer value.

Using the Subroutines

This section explains the use of the floating-point subroutines in the accelerated floating-point library, and describes the supported floating-point formats, parameter passing, return values, and fault handling. It includes a sample C program and the assembly language text generated by the compiler.

The libh subroutines must be invoked with the `call` or `callx` instructions. They cannot be invoked with the branch-and-link (`bal`) or branch-and-link-extended (`balx`) instructions.

Floating-point Formats Supported

The floating-point library supports the IEEE 754 single-precision and double-precision floating-point formats. The floating-point library also meets IEEE 754 extended-precision criteria for double-extended formats. The implemented operations fully meet the requirements of the IEEE 754 Floating-point Standard for accuracy of results and handling of special representations. In accordance with the IEEE 754 standard, all the results of libh operations are equivalent to an infinitely precise value correctly rounded to the result format. The floating-point library handles special representations such as NaNs, signed zeros and signed infinities in accordance with the IEEE 754 standard.

The floating-point library treats cases that are undefined or implementation specific in the IEEE 754 standard in the same fashion as the i960 KB processor, with one exception. While the KB processor can return a NaN value with the sign bit either cleared or set, libh always returns a NaN value with the sign bit set. Therefore, if your code must be portable across all the i960 processors, do not perform calculations that depend on the sign bit of NaN values. This practice is recommended by the IEEE 754 standard.

For detailed information on these floating-point formats and standards, see the *IEEE Standard for Binary Floating-point Arithmetic* and the *i960 KA/KB Microprocessor Programmer's Reference Manual*.

Parameter and Return Value Implementation

Parameter passing and operand configuration follow the compiler calling sequence. See your compiler user's guide for details.

The libh subroutines use source operands for parameters and destination operands for return values. The subroutines use only the global registers `g0` through `g6` for operands. They do not use literals or floating-point temporary registers. Table 6-1 indicates how libh uses specific global registers for *`src1`*, *`src2`* and *`dst`* depending on the numeric type of the value.

Table 6-1 Global Register Usage

Numeric Type	*src1*	*src2*	*dst*
extended	g0-g2	g4-g6	g0-g2
double	g0-g1	g2-g3	g0-g1
other	g0	g1	g0

For example, the `___addtf3` subroutine uses the register triplet `g0-g2` for *`src1`*, `g4-g6` for *`src2`* and `g0-g2` for *`dst`*.

The subroutine `___truncdfsf` uses the register pair `g0-g1` for *`src1`* and register `g0` for *`dst`*.

Floating-point Arithmetic Control Usage

The floating-point library uses the arithmetic control floating-point bits in the same fashion as the KB processor. See the *i960 Processor Assembler User's Guide* for information on the arithmetic control register.

The floating-point library uses the floating-point bits of the on-chip arithmetic control register for the KA processor. The CA processor does not have floating-point bits, so libh emulates them. If you are using libh with the CA processor, you must allocate a word of static memory for the emulation of the floating-point bits. To do this, include the following statement in your linker configuration file:

```
SFP_AC:
{
fpem_CA_AC =.;
} > isram
```

The compiler library subroutine `fp_setenv` writes to the floating-point arithmetic control bits. The compiler library subroutine `fp_getenv` reads the floating-point arithmetic control bits. These subroutines write to and read from the on-chip arithmetic control floating-point bits for the KA processor. They write to and read from the emulated arithmetic control floating-point bits for the CA processor. Use these subroutines instead of the `modac` instruction to access the arithmetic control floating-point bits if you want your code to be portable across all i960 processors.

Fault Handling

The floating-point library triggers the same faults, under the same circumstances, as the KB processor. As with the KB processor, all faults can be masked except for the reserved-encoding fault. With single- and double-precision floating-point values, setting the normalizing-mode bit of the floating-point arithmetic controls allows denormalized values to be used as operands for arithmetic operations, thus preventing the occurrence of reserved-encoding faults.

The floating-point library handles masked and unmasked integer-overflow faults and masked floating-point faults in the same fashion as the KB processor. Depending on the processor, libh uses either the real or emulated floating-point-fault bits of the arithmetic controls. However, libh handles unmasked floating-point faults differently as explained later in this chapter.

Code Example

Example 6-2 shows the assembly language text generated by the compiler from the C source in Example 6-1. The assembly language contains calls to the `___divdf3` and `___fixdfsi` subroutines.

Example 6-1 Sample C Program

```
#include <stdio.h>

main()
{
     int     i;
     double  d1,d2,d3;

     d2 = 12.0;
     d3 = 5.0;

     d1 = d2/d3;
     i = d1;
     printf("i=%d, d1=%f\n",i,d1);
}
```

Line 4 of Example 6-2 on the next page shows the compiler invocation command for the program. Line 16 contains the call to the `___divdf3` subroutine. Line 18 contains the call to the `___fixdfsi` subroutine.

Example 6-2 Assembly Language Generated From Sample C Program

```
#  FE version      : 1.22
#  BE version      : X5.0.317
#  Time of compilation      : Thu May 1 15:30:27 1995
#  Command line      : ic960 -S -O1 -AKA afp_ex.c
.ident      "ic960 X5.0.317 host",0x2acb250d
.file      "afp_ex.c"
.text
.align  4
.globl  _main
 _main:
 .def  _main; .val _main; .scl 2; .type 0x44; .endef
 ldl  C1,r12
 ldl  C2,g4
 movl  r12,g0
 movl  g4,g2
 callj  __divdf3
 movl  g0,r12
 callj  __fixdfsi
 mov  g0,r4
 lda  .3,g0
 mov  r4,g1
 movl  r12,g2
 b  _printf
 #Function Statistics
 # Blocks    1
 # Instructions  12
 # Instructions/Block 12
 # Loads    2
 # Stores    0
 # Calls    0
 # Registers used  r4 r12 r13 g0 g1 g2 g3 g4 g5
 #
 .def  _main; .val .; .scl -1; .endef
```

continued ☞

Example 6-2 Assembly Language Generated From Sample C Program (continued)

```
align  4
.C3:
.asciz  "i=%d, d1=%f\n"
align  3
.C2:
.word  0x00000000, 0x40140000
.align  3
.C1
.word  0x00000000, 0x40280000
```

Subroutine Reference

This section contains an entry for each function type. The entries are ordered alphabetically with the wildcard characters `?` or `*` replacing the variable portion of the function name. Each entry contains a discussion that describes how each subroutine uses operands, arithmetic controls and faults. Where necessary, the discussion describes the relationships between the source and destination operands.

___add?f3

Addition

```
___addsf3
___adddf3
___addtf3
```

Discussion

These subroutines operate as follows:

`___addsf3`	adds two single-precision floating-point values.
`___adddf3`	adds two double-precision floating-point values.
`___addtf3`	adds two extended-precision floating-point values.

The `___add?f3` subroutines perform addition as:

`src1 + src2 -> dst`

Table 6-2 shows how the `___add?f3` subroutines use global registers.

Table 6-2 ___add?f3 Global Register Usage

Subroutine	*src1*	*src2*	*dst*
___addsf3	g0(single)	g1(single)	g0(single)
___adddf3	g0-g1(double)	g2-g3(double)	g0-g1(double)
___addtf3	g0-g2(extended)	g4-g6(extended)	g0-g2(extended)

Table 6-3 shows how the `___add?f3` subroutines use the Arithmetic Control register.

Table 6-3 ___add?f3 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Rounding mode Normalizing mode
Bits set	Exception flags

Table 6-4 shows possible faults for the `___add?f3` subroutines.

Table 6-4 ___add?f3 Possible Faults

Fault	Cause
Floating reserved encoding	One or both operands denormalized and the normalizing mode bit in the arithmetic controls is not set. One or both operands are unnormals.
Floating underflow	Normalized result is too small for destination format.
Floating overflow	Result is too large for destination format.
Floating invalid operation	Operands are infinities with different signs. One or more operands is an SNaN value.
Floating inexact	Result cannot be represented exactly in destination format. Floating overflow occurred and the overflow exception was masked.

___ceil?f2

Round up to integral value

```
___ceilsf2
___ceildf2
___ceiltf2
```

Discussion

These subroutines operate as follows:

`___ceilsf2`	Single-precision round up to integral value.
`___ceildf2`	Double-precision round up to integral value.
`___ceiltf2`	Extended-precision round up to integral value.

The `___ceil?f2` subroutines convert an operand to the smallest integral floating-point value not less than `src` as:

src -> dst

Table 6-5 shows how the ___`ceil?f2` subroutines use global registers.

Table 6-5 ___ceil?f2 Global Register Usage

Subroutine	*src*	*dst*
___ceilsf2	g0(single)	g0(single)
___ceildf2	g0-g1(double)	g0-g1(double)
___ceiltf2	g0-g2(extended)	g0-g2(extended)

Table 6-6 shows how the ___`ceil?f2` subroutines use the Arithmetic Control register.

Table 6-6 ___ceil?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Rounding mode Normalizing mode
Bits set	Exception flags

Table 6-7 shows possible faults for the ___`ceil?f2` subroutines.

Table 6-7 ___ceil?f2 Possible Faults

Fault	Cause
Floating reserved encoding	Operand denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating invalid operation	Operand is an SNaN value.
Floating inexact	Operand is not an integral value.

___floor?f2

Round down to integral value

```
___floorsf2
___floordf2
___floortf2
```

Discussion

These subroutines operate as follows:

`___floorsf2`	Single-precision round down to integral value.
`___floordf2`	Double-precision round down to integral value.
`___floortf2`	Extended-precision round down to integral value.

The `___floor?f2` subroutines convert an operand to the largest integral floating-point value not greater than `src` as:

`src -> dst`

Table 6-8 shows how the `___floor?f2` subroutines use global registers.

Table 6-8 ___floor?f2 Global Register Usage

Subroutine	*src*	*dst*
___floorsf2	g0(single)	g0(single)
___floordf2	g0-g1(double)	g0-g1(double)
___floortf2	g0-g2(extended)	g0-g2(extended)

Table 6-9 shows how the `___floor?f2` subroutines use the Arithmetic Control register.

Table 6-9 ___floor?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Rounding mode Normalizing mode
Bits set	Exception flags

Table 6-10 shows possible faults for the `___floor?f2` subroutines.

Table 6-10 ___floor?f2 Possible Faults

Fault	Cause
Floating reserved encoding	Operand denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating invalid operation	Operand is an SNaN value.
Floating inexact	Operand is not an integral value.

____cls?fsi

Classify floating-point number

```
___clssfsi
___clsdfsi
___clstfsi
```

Discussion

These subroutines operate as follows:

`___clssfsi`	classifies a single-precision floating-point values.
`___clsdfsi`	classifies a double-precision floating-point values.
`___clstfsi`	classifies an extended-precision floating-point values.

The `___cls?fsi` subroutines classify floating-point values as:

src -> dst

Table 6-11 shows how the `___cls?fsi` subroutines use global registers.

Table 6-11 ___cls?fsi Global Register Usage

Subroutine	*src*	*dst*
___clssfsi	g0(single)	g0 (integer)
___clsdfsi	g0-g1(double)	g0 (integer)
___clstfsi	g0-g2(extended)	g0 (integer)

The classify operator returns an integer value indicating the result of the classification. The possible classifications and their return values are given in Table 6-12.

Table 6-12 ___cls?fsi Return Values

Classification	Return Value
Zero	s000
Denormalized number	s001
Normal finite number	s010
Infinity	s011
Quiet NaN	s100
Signaling NaN	s101
Reserved encoding	s110

Return Value is shown in binary bits, and

s is the sign bit of the value passed.

These return values are consistent with the bit patterns stored in the arithmetic-status bits of the arithmetic controls register by the i960 KB processor's `classr` and `classrl` floating-point instructions.

The classify operator does not read the arithmetic control register and does not generate any faults.

___cmp?f2

Comparison

```
___cmpsf2
___cmpdf2
___cmptf2
```

Discussion

These subroutines operate as follows:

`___cmpsf2` compares two single-precision floating-point values.

`___cmpdf2` compares two double-precision floating-point values.

`___cmptf2` compares two extended-precision floating-point values.

The `___cmp?f2` subroutines compare floating-point values as:

`src1 ? src2 -> dst`

Table 6-13 shows how the `___cmp?f2` subroutines use global registers.

Table 6-13 ___cmp?f2 Global Register Usage

Subroutine	*src1*	*src2*	*dst*
___cmpsf2	g0(single)	g1(single)	g0,AC(integer)
___cmpdf2	g0-g1(double)	g2-g3(double)	g0,AC(integer)
___cmptf2	g0-g2(extended)	g4-g6(extended)	g0,AC(integer)

The comparison operator returns an integer value indicating the result of the comparison. Table 6-14 gives the possible return values and their meanings.

Table 6-14 ___cmp?f2 Return Values

Return Value	Meaning
-1	*src1* < *src2*
0	*src1* = *src2*
1	*src1* > *src2*
3	*src1*, *src2*, or both are NaN

The ___cmp?f2 subroutines also set the condition-code flags of the Arithmetic Control register to indicate the result of the comparison. Therefore, after a comparison, your program can branch conditionally without examining the return value.

Table 6-15 shows how the ___cmp?f2 subroutines use the Arithmetic Control register.

Table 6-15 ___cmp?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Normalizing mode
Bits set	Exception flags Condition code

Table 6-16 shows possible faults for the ___cmp?f2 subroutines.

Table 6-16 ___cmp?f2 Possible Faults

Fault	Cause
Floating reserved encoding	One or both operands denormalized and the normalizing mode bit in the arithmetic controls is not set. One or both operands are unnormals.
Floating invalid operation	One or more operands is an SNaN value.

___div?f3

Division

```
___divsf3
___divdf3
___divtf3
```

Discussion

These subroutines operate as follows:

`___divsf3`	divides two single-precision floating-point values.
`___divdf3`	divides two double-precision floating-point values.
`___divtf3`	divides two extended-precision floating-point values.

The `___div?f3` subroutines perform division as:

`src1 / src2 -> dst`.

Table 6-17 shows how the `___div?f3` subroutines use global registers.

Table 6-17 ___div?f3 Global Register Usage

Subroutine	*src1*	*src2*	*dst*
___divsf3	g0(single)	g1(single)	g0(single)
___divdf3	g0-g1(double)	g2-g3(double)	g0-g1(double)
___divtf3	g0-g2(extended)	g4-g6(extended)	g0-g2(extended)

Table 6-18 shows how the `___div?f3` subroutines use the Arithmetic Control register.

Table 6-18 ___div?f3 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks
	Rounding mode
	Normalizing mode
Bits set	Exception flags

Table 6-19 shows possible faults for the `___div?f3` subroutines.

Table 6-19 ___div?f3 Possible Faults

Fault	Cause
Floating reserved encoding	One or both operands denormalized and the normalizing mode bit in the arithmetic controls is not set. One or both operands are unnormals.
Floating underflow	Result is too small for destination format.
Floating overflow	Result is too large for destination format.
Floating zero divide	The `src1` operand is 0 and the `src2` operand is numeric and finite.
Floating invalid operation	Both operands are infinities or both operands are zero. One or more operands is an SNaN value.
Floating inexact	Result cannot be represented exactly in destination format. Floating overflow occurred and the overflow exception was masked.

___extend?f?f2

Single to double conversion

```
___extenddftf2
___extendsfdf2
___extendsftf2
```

Discussion

These subroutines operate as follows:

`___extenddftf2` converts a double-precision floating-point value to an extended-precision floating-point value.

`___extendsfdf2`	converts a single-precision floating-point value to a double-precision floating-point value.
`___extendsftf2`	converts a single-precision floating-point value to an extended-precision floating-point value.

The `___extend?f?f2` subroutines perform floating-point conversion as:

`src -> dst`

Table 6-20 shows how the `___extend?f?f2` subroutines use global registers.

Table 6-20 ___extend?f?f2 Global Register Usage

Subroutine	*src*	*dst*
___extenddftf2	g0-g1(double)	g0-g2(extended)
___extendsfdf2	g0(single)	g0-g1(double)
___extendsftf2	g0(single)	g0-g2(extended)

Table 6-21 shows how the `___extend?f?f2` subroutines use the Arithmetic Control register.

Table 6-21 ___extend?f?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Normalizing mode
Bits set	Exception flags

Table 6-22 shows possible faults for the `___extend?f?f2` subroutines.

Table 6-22 ___extend?f?f2 Possible Faults

Fault	Cause
Floating reserved encoding	Operand denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating invalid operation	Source operand is an SNaN value.

___fix*

Floating-point to integer conversion with truncation

```
___fixsfsi
___fixdfsi
___fixtfsi
___fixunssfsi
___fixunsdfsi
___fixunstfsi
```

Discussion

These subroutines operate as follows:

`___fixsfsi`	converts a single-precision floating-point value to a two's-complement 32-bit integer with truncation.
`___fixdfsi`	converts a double-precision floating-point value to a two's-complement 32-bit integer with truncation.
`___fixtfsi`	converts an extended-precision floating-point value to a two's-complement 32-bit integer with truncation.

`___fixunssfsi` converts a single-precision floating-point value to an unsigned 32-bit integer with truncation.

`___fixunsdfsi` converts a double-precision floating-point value to an unsigned 32-bit integer with truncation.

`___fixunstfsi` converts an extended-precision floating-point value to an unsigned 32-bit integer with truncation.

The `___fix*` subroutines convert a floating-point value to an unsigned 32-bit integer as:

`src -> dst`

Table 6-23 shows how the `___fix*` subroutines use global registers.

Table 6-23 ___fix* Global Register Usage

Subroutine	*src*	*dst*
___fixsfsi	g0(single)	g0(integer)
___fixdfsi	g0-g1(double)	g0(integer)
___fixtfsi	g0-g2(extended)	g0(integer)
___fixunssfsi	g0(single)	g0(unsigned)
___fixunsdfsi	g0-g1(double)	g0(unsigned)
___fixunstfsi	g0-g2(extended)	g0(unsigned)

Table 6-24 shows how the `___fix*` subroutines use the Arithmetic Control register.

Table 6-24 ___fix* Arithmetic Control Usage

AC Register	Bits
Bits set	Exception flags

The following are the possible faults for the `___fix*` subroutines.

Integer overflow — Floating-point value exceeds the signed integer range (`___fix?fsi` only).

Table 6-25 shows the input values and the returned value for the `___fixuns?fsi` subroutines. Integer overflow is not signaled, however.

Table 6-25 ___fixuns?fsi Input and Return Values

Input Value Range	Returned Value
greater than or equal to 2^{32}	`0xFFFFFFFF`
from 2^{32} - 1 through -2^{32} - 1	Two's complement of the integer representing that value.
less than or equal to -2^{32}	`0`

Integer overflow is not signaled.

___float*

Integer to floating-point conversion

```
___floatsisf
___floatsidf
___floatsitf
___floatunssisf
___floatunssidf
___floatunssitf
```

Discussion

These subroutines operate as follows:

`___floatsisf`	converts a two's-complement 32-bit integer to a single-precision floating-point value.
`___floatsidf`	converts a two's-complement 32-bit integer to a double-precision floating-point value.
`___floatsitf`	converts a two's-complement 32-bit integer to an extended-precision floating-point value.
`___floatunssisf`	converts an unsigned 32-bit integer to a single-precision floating-point value.
`___floatunssidf`	converts an unsigned 32-bit integer to a double-precision floating-point value.
`___floatunssitf`	converts an unsigned 32-bit integer to an extended-precision floating-point value.

The `___float*` subroutines convert an unsigned 32-bit integer to a floating-point value as:

src -> dst

Table 6-26 shows how the `___float*` subroutines use global registers.

Table 6-26 ___float* Global Register Usage

Subroutine	*src*	*dst*
___floatsisf	g0(integer)	g0(single)
___floatsidf	g0(integer)	g0-g1(double)
___floatsitf	g0(integer)	g0-g2(extended)
___floatunssisf	g0(unsigned)	g0(single)
___floatunssidf	g0(unsigned)	g0-g1(double)
___floatunssitf	g0(unsigned)	g0-g2(extended)

Arithmetic controls are used by the ___floatsisf and ___floatunssisf subroutines only. Table 6-27 shows how the ___floatsisf and ___floatunssisf subroutines use the Arithmetic Control register.

Table 6-27 ___floatsisf and ___floatunssisf Arithmetic Control Usage

AC Register	Bits
Bits read	Rounding mode
Bits set	Exception flags

Table 6-28 shows possible faults for the ___float* subroutines.

Table 6-28 ___float* Possible Faults

Fault	Cause
Floating inexact	Result cannot be represented exactly in destination format.

___logb?f2

Extract unbiased exponent

```
___logbsf2
___logbdf2
___logbtf2
```

Discussion

These subroutines operate as follows:

`___logbsf2`	extracts an unbiased single-precision exponent.
`___logbdf2`	extracts an unbiased double-precision exponent.
`___logbtf2`	extracts an unbiased extended-precision exponent.

The `___logb?f2` subroutines extract an unbiased exponent as:

`src -> dst`

Table 6-29 shows how the `___logb?f2` subroutines use global registers.

Table 6-29 ___logb?f2 Global Register Usage

Subroutine	*src*	*dst*
___logbsf2	g0(single)	g0(single)
___logbdf2	g0-g1(double)	g0-g1(double)
___logbtf2	g0-g2(extended)	g0-g2(extended)

Table 6-30 shows how the `___logb?f2` subroutines use the Arithmetic Control register.

Table 6-30 ___logb?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Normalizing mode
Bits set	Exception flags

Table 6-31 shows possible faults for the `___logb?f2` subroutines.

Table 6-31 ___logb?f2 Possible Faults

Fault	Cause
Floating reserved encoding	Operand denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating invalid operation	Operands are infinities with different signs. One or more operands is an SNaN value.
Floating zero divide	Operand is 0.

___mul?f3

Multiplication

```
___mulsf3
___muldf3
___multf3
```

Discussion

These subroutines operate as follows:

`___mulsf3`	multiplies two single-precision floating-point values.
`___muldf3`	multiplies two double-precision floating-point values.
`___multf3`	multiplies two extended-precision floating-point values.

The `___mul?f3` subroutines perform multiplication as:

`src1 * src2 -> dst`

Table 6-32 shows how the `___mul?f3` subroutines use global registers.

Table 6-32 ___mul?f3 Global Register Usage

Subroutine	*src1*	*src2*	*dst*
___mulsf3	g0(single)	g1(single)	g0(single)
___muldf3	g0-g1(double)	g2-g3(double)	g0-g1(double)
___multf3	g0-g2(extended)	g4-g6(extended)	g0-g2(extended)

Table 6-33 shows how the ___`mul?f3` subroutines use the Arithmetic Control register.

Table 6-33 ___mul?f3 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Rounding mode Normalizing mode
Bits set	Exception flags

Table 6-34 shows possible faults for the ___`mul?f3` subroutines.

Table 6-34 ___mul?f3 Possible Faults

Fault	Cause
Floating reserved encoding	One or both operands denormalized and the normalizing mode bit in the arithmetic controls is not set. One or both operands are unnormals.
Floating underflow	Normalized result is too small for destination format.
Floating overflow	Result is too large for destination format.
Floating invalid operation	One operand is 0 and the other operand is infinity. One or more operands is an SNaN value.
Floating inexact	Result cannot be represented exactly in destination format. Floating overflow occurred and the overflow exception was masked.

___rem?f3

Remaindering

```
___remsf3
___remdf3
___remtf3
```

Discussion

These subroutines implement the KB `remr` instruction. They operate as follows:

`___remsf3` returns a single-precision KB remainder.

`___remdf3` returns a double-precision KB remainder.

`___remtf3` returns an extended-precision KB remainder.

The `___rem?f3` subroutines perform remaindering as:

src1 <rem> *src2* -> *dst*

Table 6-35 shows how the `___rem?f3` subroutines use global registers.

Table 6-35 ___rem?f3 Global Register Usage

Subroutine	*src1*	*src2*	*dst*	*dst2*
___remsf3	g0(single)	g1(single)	g0(single)	g1(integer)
___remdf3	g0-g1 (double)	g2-g3 (double)	g0-g1 (double)	g2(integer)
___remtf3	g0-g2 (extended)	g4-g6 (extended)	g0-g2 (extended)	g4(integer)

The ___rem?f3 subroutines offer assembly language access to an integer return value as shown under *dst2* in Table 6-35. The upper 28 bits of this integer value are set to zero, while the four low order bits match the arithmetic status field bits of the KB remr instruction. Table 6-36 shows the possible integer return values and their meanings.

Table 6-36 ___rem?f3 Integer Return Values

Return Value	Meaning
0	QS, set if the remainder after the QR reduction would be non-zero (the "sticky" bit of the quotient)
1	QR, the value the next quotient bit would have if one more reduction were performed (the "round" bit of the quotient)
2	Q0, the last quotient bit
3	Q1, the next-to-last quotient bit

Table 6-37 shows how the ___rem?f3 subroutines use the Arithmetic Control register.

Table 6-37 ___rem?f3 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Normalizing mode
Bits set	Exception flags

Table 6-38 shows possible faults for the ___rem?f3 subroutines.

Table 6-38 ___rem?f6 Possible Faults

Fault	Cause
Floating reserved encoding	One or both operands denormalized and the normalizing mode bit in the arithmetic controls is not set. One or both operands are unnormals.
Floating invalid operation	*`src1`* is infinite and/or *`src2`* is 0. One or more operands is an SNaN value.

___rint?f2

Round to nearest integral value

```
___rintsf2
___rintdf2
___rinttf2
```

Discussion

These subroutines operate as follows:

`___rintsf2`	Single-precision round to nearest integral value.
`___rintdf2`	Double-precision round to nearest integral value.
`___rinttf2`	Extended-precision round to nearest integral value.

The `___rint?f2` subroutines perform rounding as:

`src -> dst`

Table 6-39 shows how the `___rint?f2` subroutines use global registers.

Table 6-39 ___rint?f2 Global Register Usage

Subroutine	*src*	*dst*
___rintsf2	g0(single)	g0(single)
___rintdf2	g0-g1(double)	g0-g1(double)
___rinttf2	g0-g2(extended)	g0-g2(extended)

Table 6-40 shows how the `___rint?f2` subroutines use the Arithmetic Control register.

Table 6-40 ___rint?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Normalizing mode
Bits set	Exception flags

Table 6-41 shows possible faults for the `___rint?f2` subroutines.

Table 6-41 ___rint?f2 Possible Faults

Fault	Cause
Floating reserved encoding	Operand denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating invalid operation	Operand is an SNaN value.
Floating inexact	Operand is not an integral value.

___rmd?f3

IEEE Remaindering

```
___rmdsf3
___rmddf3
___rmdtf3
```

Discussion

These subroutines perform IEEE 754 remaindering as follows:

`___rmdsf3`	returns a single-precision IEEE remainder.
`___rmddf3`	returns a double-precision IEEE remainder.
`___rmdtf3`	returns an extended-precision IEEE remainder.

The `___rmd?f3` subroutines perform IEEE 754 remaindering as:

`src1 <rmd> src2 -> dst`

Table 6-42 shows how the `___rmd?f3` subroutines use global registers.

Table 6-42 ___rmd?f3 Global Register Usage

Subroutine	*src1*	*src2*	*dst*	*dst2*
___rmdsf3	g0(single)	g1(single)	g0(single)	g1(unsigned)
___rmddf3	g0-g1 (double)	g2-g3 (double)	g0-g1 (double)	g2(unsigned)
___rmdtf3	g0-g2 (extended)	g4-g6 (extended)	g0-g2 (extended)	g4(unsigned)

The `___rmd?f3` subroutines offer assembly language access to an unsigned integer return value as shown under *`dst2`* in Table 6-42. This integer return value is comprised of the least significant 32 bits of the magnitude of the integral quotient, rounded per the IEEE remaindering operation.

Table 6-43 shows how the ___rmd?f3 subroutines use the Arithmetic Control register.

Table 6-43 ___rmd?f3 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Normalizing mode
Bits set	Exception flags

Table 6-44 shows possible faults for the ___rmd?f3 subroutines.

Table 6-44 ___rmd?f3 Possible Faults

Fault	Cause
Floating reserved encoding	One or both operands denormalized and the normalizing mode bit in the arithmetic controls is not set. One or both operands are unnormals.
Floating invalid operation	*src1* is infinite and/or *src2* is zero. One or more operands is an SNaN value.

___round?f2

Round to integral value

```
___roundsf2
___rounddf2
___roundtf2
```

Discussion

These subroutines operate as follows:

`___roundsf2`	Single-precision round to integral value.
`___rounddf2`	Double-precision round to integral value.
`___roundtf2`	Extended-precision round to integral value.

The `___round?f2` subroutines convert an operand to an integral floating-point value as:

`src -> dst`

Table 6-45 shows how the `___round?f2` subroutines use global registers.

Table 6-45 ___round?f2 Global Register Usage

Subroutine	*src*	*dst*
___roundsf2	g0(single)	g0(single)
___rounddf2	g0-g1(double)	g0-g1(double)
___roundtf2	g0-g2(extended)	g0-g2(extended)

Table 6-46 shows how the `___round?f2` subroutines use the Arithmetic Control register.

Table 6-46 ___round?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Rounding mode Normalizing mode
Bits set	Exception flags

Table 6-47 shows possible faults for the `___round?f2` subroutines.

Table 6-47 ___round?f2 Possible Faults

Fault	Cause
Floating reserved encoding	Operand denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating invalid operation	Operand is an SNaN value.
Floating inexact	Operand is not an integral value.

___round?fsi

Floating-point to integer conversion with rounding

```
___roundsfsi
___rounddfsi
___roundtfsi
```

Discussion

These subroutines operate as follows:

`___roundsfsi`	converts a single-precision floating-point value to a two's-complement 32-bit integer.
`___rounddfsi`	converts a double-precision floating-point value to a two's-complement 32-bit integer.
`___roundtfsi`	converts an extended-precision floating-point value to a two's-complement 32-bit integer.

The `___round?fsi` subroutines round the results according to the integer type of the destination operand and the setting of the rounding-mode flags of the floating-point arithmetic controls. They perform conversions as:

`src -> dst`

Table 6-48 shows how the `___round?fsi` subroutines use global registers.

Table 6-48 ___round?fsi Global Register Usage

Subroutine	*src*	*dst*
___roundsfsi	g0(single)	g0(integer)
___rounddfsi	g0-g1(double)	g0(integer)
___roundtfsi	g0-g2(extended)	g0(integer)

Table 6-49 shows how the `___round?fsi` subroutines use the Arithmetic Control register.

Table 6-49 ___round?fsi Arithmetic Control Usage

AC Register	Bits
Bits read	Rounding mode
Bits set	Integer overflow flag

Table 6-50 shows possible faults for the `___round?fsi` subroutines.

Table 6-50 ___round?fsi Possible Faults

Fault	Cause
Integer overflow	Floating-point value exceeds the signed integer range.

___rounduns?fsi

Floating-point to unsigned integer conversion with rounding

```
___roundunssfsi
___roundunsdfsi
___roundunstfsi
```

Discussion

These subroutines operate as follows:

`___roundunssfsi`	converts a single-precision floating-point value to an unsigned 32-bit integer.
`___roundunsdfsi`	converts a double-precision floating-point value to an unsigned 32-bit integer.
`___roundunstfsi`	converts an extended-precision floating-point value to an unsigned 32-bit integer.

The `___rounduns?fsi` subroutines round the results according to the integer type of the destination operand and the setting of the rounding-mode flags of the floating-point arithmetic controls. They perform conversions as:

`src -> dst`

Table 6-51 shows how the `___rounduns?fsi` subroutines use global registers.

Table 6-51 ___rounduns?fsi Global Register Usage

Subroutine	*src*	*dst*
___roundunssfsi	g0(single)	g0(unsigned)
___roundunsdfsi	g0-g1(double)	g0(unsigned)
___roundunstfsi	g0-g2(extended)	g0(unsigned)

Table 6-52 shows how the `___rounduns?fsi` subroutines use the Arithmetic Control register.

Table 6-52 ___rounduns?fsi Arithmetic Control Usage

AC Register	Bits
Bits read	Rounding mode

The `___rounduns?fsi` subroutines return the hexadecimal value 0xFFFFFFFF when the result is too large to be represented as an unsigned 32-bit integer. Integer overflow is not signaled, however.

___scale?fsi?f

Scale floating-point value by signed integer value

```
___scalesfsisf
___scaledfsidf
___scaletfsitf
```

Discussion

These subroutines operate as follows:

`___scalesfsisf`	scales a single-precision floating-point value.
`___scaledfsidf`	scales a double-precision floating-point value.
`___scaletfsitf`	scales an extended-precision floating-point value.

The `___scale?fsi?f` subroutines scale the source floating-point value by the signed 32-bit integer operand as:

$src1 * 2^{src2} \rightarrow dst.$

Since they have operands of different types, the `___scale?fsi?f` subroutines may require special handling in user-supplied fault handlers, as described later in this chapter. The first operand is always a floating-point value, and the second is always a signed integer.

Table 6-53 shows how the `___scale?fsi?f` subroutines use global registers.

Table 6-53 ___scale?fsi?f Global Register Usage

Subroutine	*src1*	*src2*	*dst*
___scalesfsisf	g0(single)	g1(integer)	g0(single)
___scaledfsidf	g0-g1(double)	g2(integer)	g0-g1(double)
___scaletfsitf	g0-g2(extended)	g4(integer)	g0-g2(extended)

Table 6-54 shows how the `___scale?fsi?f` subroutines use the Arithmetic Control register.

Table 6-54 ___scale?fsi?f Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Normalizing mode
Bits set	Exception flags

Table 6-55 shows possible faults for the `___scale?fsi?f` subroutines.

Table 6-55 ___scale?fsi?f Possible Faults

Fault	Cause
Floating reserved encoding	Operand is denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating underflow	Normalized result is too small for destination format.
Floating overflow	Result is too large for destination format.
Floating invalid operation	Operand is an SNaN value.
Floating inexact	Floating overflow occurred and the overflow exception was masked.

___sub?f3

Subtraction

```
___subsf3
___subdf3
___subtf3
```

Discussion

These subroutines operate as follows:

`___subsf3`	subtracts two single-precision floating-point values.
`___subdf3`	subtracts two double-precision floating-point values.
`___subtf3`	subtracts two extended-precision floating-point values.

The `___sub?f3` subroutines perform subtraction as:

`src1 - src2 -> dst`

Table 6-56 shows how the `___sub?f3` subroutines use global registers.

Table 6-56 ___sub?f3 Global Register Usage

Subroutine	*src1*	*src2*	*dst*
___subsf3	g0(single)	g1(single)	g0(single)
___subdf3	g0-g1(double)	g2-g3(double)	g0-g1(double)
___subtf3	g0-g2(extended)	g4-g6(extended)	g0-g2(extended)

Table 6-57 shows how the `___sub?f3` subroutines use the Arithmetic Control register.

Table 6-57 ___sub?f3 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Rounding mode Normalizing mode
Bits set	Exception flags

Table 6-58 shows possible faults for the `___sub?f3` subroutines.

Table 6-58 ___sub?f3 Possible Faults

Fault	Cause
Floating reserved encoding	One or both operands denormalized and the normalizing mode bit in the arithmetic controls is not set. One or both operands are unnormals.
Floating underflow	Normalized result is too small for destination format.
Floating overflow	Result is too large for destination format.
Floating invalid operation	Operands are infinities of like signs. One or more operands is an SNaN value.
Floating inexact	Result cannot be represented exactly in destination format. Floating overflow occurred and the overflow exception was masked.

___trunc?f?f2

Double to single conversion

```
___truncdfsf2
___trunctfdf2
___trunctfsf2
```

Discussion

These subroutines operate as follows:

`___truncdfsf2`	converts a double-precision floating-point value to a single-precision floating-point value.
`___trunctfdf2`	converts an extended-precision floating-point value to a double-precision floating-point value.
`___trunctfsf2`	converts an extended-precision floating-point value to a single-precision floating-point value.

The `___trunc?f?f2` subroutines round the results according to the setting of the rounding-mode flags of the floating-point arithmetic controls. They perform floating-point format conversions as:

`src -> dst`

Table 6-59 shows how the `___trunc?f?f2` subroutines use global registers.

Table 6-59 ___trunc?f?f2 Global Register Usage

Subroutine	*src*	*dst*
___truncdfsf2	g0-g1(double)	g0(single)
___trunctfdf2	g0-g2(extended)	g0-g1(double)
___trunctfsf2	g0-g2(extended)	g0(single)

Table 6-60 shows how the `___trunc?f?f2` subroutines use the Arithmetic Control register.

Table 6-60 ___trunc?f?f2 Arithmetic Control Usage

AC Register	Bits
Bits read	Floating-point exception masks Rounding mode Normalizing mode
Bits set	Exception flags

Table 6-61 shows possible faults for the `___trunc?f?f2` subroutines.

Table 6-61 Faults for __trunc?f?f2

Fault	Cause
Floating reserved encoding	Operand denormalized and the normalizing mode bit in the arithmetic controls is not set. Operand is unnormal.
Floating underflow	Result is too small for destination format.
Floating overflow	Result is too large for destination format.
Floating invalid operation	Source operand is an SNaN value.
Floating inexact	Result cannot be represented exactly in destination format. Floating overflow occurred and the overflow exception was masked.

Unmasked Floating-point Fault Handling

This section describes the way that the floating-point library handles unmasked floating-point faults and tells you how to create custom unmasked fault-handling subroutines.

The libh libraries contain eighteen unmasked fault-handling subroutines. Three subroutines are available for each of the six floating-point faults. The floating-point faults are: inexact result, invalid operation, overflow, reserved encoding, underflow, and zero divide. For each of these faults, the libh library provides a subroutine for single-precision operations, a subroutine for double-precision operations and a subroutine for extended-precision operations.

When a floating-point fault occurs during the execution of a libh subroutine, and the specific fault is unmasked in the arithmetic controls, control is transferred to the appropriate fault-handling subroutine. Parameter passing and operand configuration follow the compiler calling sequence. See your compiler user's guide for details.

The default libh fault-handling subroutines return values and take no action. These subroutines are not intended for use by any application, serving only as placeholders for user-supplied fault-handling subroutines.

You can create custom fault-handling subroutines by writing C subroutines based on the prototype declarations of the fault-handling subroutines contained in the floating-point libraries. When the program is linked, the linker uses your version of the subroutines in place of the subroutines in the libh libraries.

The rest of this section describes the prototype declarations for the fault-handling subroutines and describes the actions of the fault-handling subroutines contained in the libh libraries. The `src` subdirectory under the `I960BASE` or `G960BASE` directory contains example source code for fault-handling subroutines if you have installed source.

See the *i960 KA/KB Microprocessor Programmer's Reference Manual* for more information on fault handling and floating-point faults.

Parameters

The fault-handling subroutines take either two or three parameters, depending on whether the fault is detected before or after the operation of the faulting subroutine.

The floating-point subroutines allow handling from underflow, overflow and inexact-result faults after the operation of the faulting subroutine. The fault-handling subroutines for these faults take two parameters. The first parameter, named *`result`*, is the properly rounded *`dst`* operand from the faulting subroutine. In the case of underflow or overflow faults, the *`result`* parameter is scaled to make it representable in the floating-point format of the subroutine.

Additional libh subroutines handle reserved-encoding, invalid-operation and zero-divide faults before the operation of the faulting subroutine. The single- and double-precision fault-handling subroutines for these faults take three parameters. The extended-precision subroutines take two parameters, as described at the end of this section. The first two parameters in both cases are named *`src1`* and *`src2`*. They are the *`src1`* and *`src2`* operands from the faulting subroutine.

The last parameter for all the fault-handling subroutines is named *opcode*. This parameter is an integer value that indicates the operation of the faulting subroutine. Using this indicator, your fault-handling subroutine can branch conditionally on the operation of the calling floating-point subroutine. Table 6-62 shows the possible values for the *opcode* parameter, in decimal, and their operations.

Table 6-62 Possible Values for the *opcode* Parameter

Opcode Value	Operation
1	___add?f3 or ___sub?f3
2	___div?f3
3	___mul?f3
4	___floatsisf
5	___floatunssisf
6	___trunctfdf2
7	___extenddftf2
8	___trunctfsf2
9	___extendsftf2
10	___truncdfsf2
11	___extendsfdf2
12	___cmp?f2
13	___scale?fsi?f
14	___logb?f2
15	___rem?f3
16	___rint?f2
17	___rmd?f3
18	___round?f2
19	___ceil?f2
20	___floor?f2

Thus, the single-precision subroutine prototype for the inexact-result fault is as follows:

```
float AFP_Fault_Inexact_S(float result, int opcode);
```

result is the properly rounded *dst* operand from the faulting subroutine.

opcode is an integer value indicating the operation of the faulting subroutine.

The double-precision subroutine prototype for the invalid-operation fault is as follows:

```
double AFP_Fault_Invalid_Operation_D(double src1, double
src2, int opcode);
```

src1 is the *src1* operand from the faulting subroutine.

src2 is the *src2* operand from the faulting subroutine.

opcode is an integer value indicating the operation of the faulting subroutine.

The extended-precision subroutines for faults that occur before the operation take two parameters rather than three. These subroutines pack both the *src2* operand from the faulting subroutine and the *opcode* value into a single union construct named *src2*. This packing optimizes global register usage. Example 6-3 shows how the union construct is defined.

Example 6-3 Union Definition

```
union fild {
     struct {
          int w1, w2, w3, op;
     } f1;
     long double f2;
}
```

The *f2* field contains the *src2* operand from the faulting subroutine. The *f1.op* field contains the *opcode* value.

Therefore, the extended-precision subroutine prototype for the invalid-operation fault is as follows:

```
long double AFP_Fault_Invalid_Operation_D(long double
src1, union fild src2);
```

src1	is the *src1* operand from the faulting subroutine.
src2.f2	is the *src2* operand from the faulting subroutine.
src2.f1.op	is the *opcode* value.

Return Values

The faulting subroutine returns the return value from the fault-handling subroutines.

The fault-handling subroutines provided with the floating-point libraries return the value zero for faults detected prior to the floating-point operation and return the *result* parameter for faults detected after the operation.

Fault-handling Subroutines

The following sections describe each of the available subroutines.

Inexact Result

The prototype declarations for the inexact-result fault-handling subroutines are:

```
float AFP_Fault_Inexact_S(float result, int opcode);
double AFP_Fault_Inexact_D(double result, int opcode);
long double AFP_Fault_Inexact_T(long double result, int
opcode);
```

result	is the properly rounded *dst* operand from the faulting subroutine.
opcode	is an integer value indicating the operation of the faulting subroutine. See the Parameters section for the possible values for the *opcode* parameter and their meanings.

The default subroutines supplied with libh return the *result* parameter.

Invalid Operation

The prototype declarations for the invalid-operation fault-handling subroutines are:

```
float AFP_Fault_Invalid_Operation_S(float src1, float
src2, int opcode);
```

```
double AFP_Fault_Invalid_Operation_D(double src1, double
src2, int opcode);
```

```
long double AFP_Fault_Invalid_Operation_T(long double
src1, union fild src2);
```

src1	is the *src1* operand from the faulting subroutine.
src2	is the *src2* operand from the faulting subroutine. For the `AFP_Fault_Invalid_Operation_T` subroutine, this value is in *src2.f2*.
opcode	is an integer value indicating the operation of the faulting subroutine. For the `AFP_Fault_Invalid_Operation_T` subroutine, this value is in *src2.f1.op*. See the Parameters section for the possible values for the *opcode* parameter and their meanings.

See the Parameters section for an explanation of the `fild` union.

When any of the subroutines listed below result in an invalid-operation fault, the *src1* operand must be an SNaN. Do not reference the *src2* operand when dealing with an invalid-operation fault resulting from these subroutines:

```
___ceil?f2
___floor?f2
___extend?f?f2
___logb?f2
___rint?f2
___round?f2
___scale?fsi?f
___trunc?f?f2
```

The default subroutines supplied with libh return the value zero.

Overflow

The prototype declarations for the overflow fault-handling subroutines are:

```
float AFP_Fault_Overflow_S(float result, int opcode);
```

```
double AFP_Fault_Overflow_D(double result, int opcode);
```

```
long double AFP_Fault_Overflow_T(long double result, int
opcode);
```

result	is the properly rounded *dst* operand from the faulting subroutine scaled by 2^{-192} for single-precision operations, 2^{-1536} for double-precision operations and 2^{-24576} for extended-precision operations. If massive overflow occurs, the *result* parameter is the properly signed infinity.
opcode	is an integer value indicating the operation of the faulting subroutine. See the Parameters section for the possible values for the *opcode* parameter and their meanings.

The `___scale?fsi?f` and `trunc?f?f2` subroutines may produce results massively exceeding the representable range of the *result* parameter's floating-point format. If the exponent adjustment described above does not bring the value within representable range, an infinity of the proper sign is used.

This subroutine receives a single value which is the properly rounded result after scaling of the faulting operation. When the overflow exception is masked, either a properly signed infinity or a maximum magnitude finite number (depending on the current rounding mode) is returned and the overflow flag bit in the Arithmetic Controls register is set. The default subroutine supplied with libh returns the *result* parameter.

Reserved Encoding

The prototype declarations for the reserved-encoding fault-handling subroutines are:

```
float AFP_Fault_Reserved_Encoding_S(float src1, float
src2, int opcode);
```

```
double AFP_Fault_Reserved_Encoding_D(double src1, double
src2, int opcode);
```

```
long double AFP_Fault_Reserved_Encoding_T(long double
src1, union fild src2);
```

src1	is the *src1* operand from the faulting subroutine.
src2	is the *src2* operand from the faulting subroutine. For the `AFP_Fault_Reserved_Encoding_T` subroutine, this value is in *src2.f2*.
opcode	is an integer value indicating the operation of the faulting subroutine. For the `AFP_Fault_Reserved_Encoding_T` subroutine, this value is in *src2.f1.op*. See the Parameters section for the possible values for the *opcode* parameter and their meanings.

See the Parameters section for an explanation of the `fild` union.

When any of the operations listed below result in a reserved-encoding fault, the *src1* operand must be the denormal or unnormal value which caused the fault. Do not reference the *src2* operand when dealing with a reserved-encoding fault resulting from these operations:

```
___extend?f?f2
___logb?f2
___rint?f2
___round?f2
___scale?fsi?f
___trunc?f?f2
```

The default subroutines supplied with libh return the value zero.

NOTE. *Reserved-encoding faults cannot be masked. However, setting the normalizing-mode bit of the floating-point arithmetic controls prevents reserved-encoding faults with single- and double-precision values. This action permits denormalized values to be used as operands for arithmetic operations.*

Underflow

The prototype declarations for the underflow fault-handling subroutines are:

```
float AFP_Fault_Underflow_S(float result, int opcode);
```

```
double AFP_Fault_Underflow_D(double result, int opcode);
```

```
long double AFP_Fault_Underflow_T(long double result, int
opcode);
```

`result` — is the properly rounded *`dst`* operand from the faulting subroutine scaled by 2^{192} for single-precision operations, 2^{1536} for double-precision operations and 2^{24576} for extended-precision operations. If massive underflow occurs, the *`result`* parameter is the properly signed zero.

`opcode` — is an integer value indicating the operation of the faulting subroutine. See the Parameters section for the possible values for the *`opcode`* parameter and their meanings.

This subroutine receives a single value which is the properly rounded result after scaling of the faulting subroutine. When the underflow exception is masked, either a properly signed zero or a denormalized number (depending on the magnitude of the result) is returned and the underflow flag bit in the Arithmetic Controls register is set. The default subroutine supplied with libh returns the scaled value.

Zero Divide

The prototype declarations for the zero-divide fault-handling subroutines are:

```
float AFP_Fault_Zero_Divide_S(float src1, float src2, int
opcode);
```

```
double AFP_Fault_Zero_Divide_D(double src1, double src2,
int opcode);
```

```
long double AFP_Fault_Zero_Divide_T(long double src1,
union fild src2);
```

src1 is the *src1* operand from the faulting subroutine. *src1* must be a finite non-zero value.

src2 is the *src2* operand from the faulting subroutine. *src2* must be a signed zero value. For the `AFP_Fault_Zero_Divide_T` subroutine, this value is in *src2.f2*.

opcode is an integer value indicating the operation of the faulting subroutine. *opcode* must be the value `2` for division. For the `AFP_Fault_Zero_Divide_T` subroutine, this value is in *src2.f1.op*. See the Parameters section for the possible values for the *opcode* parameter and their meanings.

See the Parameters section for an explanation of the `fild` union.

The `___scale?fsi?f` and `___logb?f2` subroutines signal a zero-divide when the *src1* operand is zero. Do not reference the *src2* operand when dealing with a zero-divide fault resulting from a `___scale?fsi?f` or `___logb?f2` operation.

Function Interdependencies

A

High-level functions often refer to low-level functions. Table A-1 shows which low-level functions are required by each high-level function. If you are retargeting your application to run in other than a directly supported environment, you must rewrite the functions shown in the right column. These functions are described in Chapter 5 or in *C: A Reference Manual*.

Table A-1 Cross-reference of low-level functions

This high-level function:	Depends on these low-level functions:	
_exit_init	_errno_ptr,	_semaphore_init
	_exit_create,	
_HL_init	_arg_init,	_sig_int_dfl,
	_err_no_ptr,	_sig_null,
	_exit_create,	_sig_read_dfl,
	_exit_ptr,	_sig_segv_dfl,
	_LL_init,	_sig_term_dfl,
	_semaphore_init,	_sig_write_dfl,
	_semaphore_signal,	_stdio_create,
	_semaphore_wait,	_stdio_ptr,
	_sig_abrt_dfl,	_stdio_stdopen,
	_sig_alloc_dfl,	_thread_create,
	_sig_fpe_dfl,	isatty,
	_sig_ill_dfl,	sbrk
	_sig_free_dfl,	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
_stdio_init	_err_no_ptr,	_sig_null,
	_exit_ptr,	_sig_read_dfl,
	_semaphore_init,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	_stdio_create,
	_sig_alloc_dfl,	_stdio_ptr,
	_sig_fpe_dfl,	_stdio_stdopen,
	_sig_free_dfl,	isatty,
	_sig_ill_dfl,	sbrk
	_sig_int_dfl,	
abort	_err_no_ptr,	_sig_int_dfl,
	_exit,	_sig_null,
	_sig_abrt_dfl,	_sig_read_dfl,
	_sig_alloc_dfl,	_sig_segv_dfl,
	_sig_fpe_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl,
	_sig_ill_dfl,	
acos	_err_no_ptr	
asctime	_err_no_ptr	
asin	_err_no_ptr	
assert	_err_no_ptr,	_sig_null,
	_exit,	_sig_read_dfl,
	_exit_ptr,	_sig_segv_dfl,
	_map_length,	

continued ☞

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
assert (continued)	_semaphore_signal,	_sig_int_dfl,
	_semaphore_wait,	_sig_term_dfl,
	_sig_abrt_dfl,	_sig_write_dfl,
	_sig_alloc_dfl,	_stdio_ptr,
	_sig_fpe_dfl,	lseek,
	_sig_free_dfl,	write
	_sig_ill_dfl,	
atan	_err_no_ptr	
atan2	_err_no_ptr	
atan2f	_err_no_ptr	
atan2l	_thread_ptr	
atanf	_err_no_ptr	
atanl	_thread_ptr	
atexit	_exit_ptr,	_semaphore_wait
	_semaphore_signal,	
atof	_err_no_ptr	
atol	_err_no_ptr	
calloc	_err_no_ptr,	_sig_null,
	_sig_abrt_dfl,	_sig_read_dfl,
	_sig_alloc_dfl,	_sig_segv_dfl,
	_sig_fpe_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl,
	_sig_ill_dfl,	sbrk
	_sig_int_dfl,	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
clearerr	_semaphore_signal,	_semaphore_wait
cosh	_err_no_ptr,	
ctime	_err_no_ptr,	_sig_null,
	_sig_abrt_dfl,	_sig_read_dfl,
	_sig_alloc_dfl,	_sig_segv_dfl,
	_sig_fpe_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl,
	_sig_ill_dfl,	_tzset_ptr,
	_sig_int_dfl,	sbrk
div	_err_no_ptr,	
exit	_err_no_ptr,	_sig_null,
	_exit,	_sig_read_dfl,
	_exit_ptr,	_sig_segv_dfl,
	_map_length,	_sig_term_dfl,
	_semaphore_delete,	_sig_write_dfl,
	_semaphore_signal,	_stdio_ptr,
	_semaphore_wait,	_thread_ptr,
	_sig_abrt_dfl,	c_term,
	_sig_alloc_dfl,	close,
	_sig_fpe_dfl,	lseek,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	unlink,
	_sig_int_dfl,	write
exp	_err_no_ptr	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
expf	_err_no_ptr	
fclose	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_delete,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	_stdio_ptr,
	_sig_alloc_dfl,	close,
	_sig_fpe_dfl,	lseek,
	_sig_free_dfl,	unlink,
	_sig_ill_dfl,	write
fcloseall	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_delete,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	_stdio_ptr,
	_sig_alloc_dfl,	close,
	_sig_fpe_dfl,	lseek,
	_sig_free_dfl,	unlink,
	_sig_ill_dfl,	write

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
fdopen	_err_no_ptr,	_sig_ill_dfl,
	_exit_ptr,	_sig_int_dfl,
	_semaphore_init,	_sig_null,
	_semaphore_signal,	_sig_read_dfl,
	_semaphore_wait,	_sig_segv_dfl,
	_sig_abrt_dfl,	_sig_term_dfl,
	_sig_alloc_dfl,	_sig_write_dfl,
	_sig_fpe_dfl,	isatty,
	_sig_free_dfl,	sbrk
fflush	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	
fgetc	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_signal,	_sig_segv_dfl,
	_semaphore_wait,	_sig_term_dfl,
	_sig_abrt_dfl,	_sig_write_dfl,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
fgetchar	_err_no_ptr,	_sig_null,
	_exit_ptr,	_sig_read_dfl,
	_map_length,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	_stdio_ptr,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
	_sig_int_dfl,	
fgetpos	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	
fgets	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_signal,	_sig_segv_dfl,
	_semaphore_wait,	_sig_term_dfl,
	_sig_abrt_dfl,	_sig_write_dfl,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
fileno	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	
flushall	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	
fopen	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_semaphore_init,	_sig_read_dfl,
	_semaphore_signal,	_sig_segv_dfl,
	_semaphore_wait,	_sig_term_dfl,
	_sig_abrt_dfl,	_sig_write_dfl,
	_sig_alloc_dfl,	close,
	_sig_fpe_dfl,	isatty,
	_sig_free_dfl,	open,
	_sig_ill_dfl,	sbrk
fprintf	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	

continued ☞

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
fputc	_err_no_ptr,	_sig_int_dfl,
	_semaphore_signal,	_sig_null,
	_semaphore_wait,	_sig_read_dfl,
	_sig_abrt_dfl,	_sig_segv_dfl,
	_sig_alloc_dfl,	_sig_term_dfl,
	_sig_fpe_dfl,	_sig_write_dfl,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
fputchar	_err_no_ptr,	_sig_null,
	_semaphore_signal,	_sig_read_dfl,
	_semaphore_wait,	_sig_segv_dfl,
	_sig_abrt_dfl,	_sig_term_dfl,
	_sig_alloc_dfl,	_sig_write_dfl,
	_sig_fpe_dfl,	_stdio_ptr,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
	_sig_int_dfl,	
fputs	_err_no_ptr,	_sig_fpe_dfl,
	_semaphore_signal,	_sig_null,
	_semaphore_wait,	_sig_read_dfl,
	_sig_abrt_dfl,	_sig_segv_dfl,
	_sig_alloc_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl,
	_sig_ill_dfl,	sbrk,
	_sig_int_dfl,	write

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
fread	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_signal,	_sig_segv_dfl,
	_semaphore_wait,	_sig_term_dfl,
	_sig_abrt_dfl,	_sig_write_dfl,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
free	_err_no_ptr,	_sig_int_dfl,
	_sig_abrt_dfl,	_sig_null,
	_sig_alloc_dfl,	_sig_read_dfl,
	_sig_fpe_dfl,	_sig_segv_dfl,
	_sig_free_dfl,	_sig_term_dfl,
	_sig_ill_dfl,	_sig_write_dfl,
freopen	_err_no_ptr,	close,
	_exit_ptr,	isatty,
	_map_length,	lseek,
	_semaphore_init,	open,
	_semaphore_signal,	unlink,
	_semaphore_wait,	write
fscanf	_err_no_ptr,	_semaphore_wait
	_semaphore_signal,	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
fseek	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	
fsetpos	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	
ftell	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	
fwrite	_err_no_ptr,	_sig_ill_dfl,
	_semaphore_signal,	_sig_int_dfl,
	_semaphore_wait,	_sig_null,
	_sig_abrt_dfl,	_sig_read_dfl,
	_sig_alloc_dfl,	_sig_segv_dfl
	_sig_fpe_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl
getc	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_signal,	_sig_segv_dfl,
	_semaphore_wait,	_sig_term_dfl,
	_sig_abrt_dfl,	_sig_write_dfl,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,

continued ☞

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	**Depends on these low-level functions:**	
getc (continued)	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
getchar	_err_no_ptr,	_sig_null,
	_exit_ptr,	_sig_read_dfl,
	_map_length,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	_stdio_ptr,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
	_sig_int_dfl,	
getopt	_err_no_ptr,	write
gets	_err_no_ptr,	_sig_null,
	_exit_ptr,	_sig_read_dfl,
	_map_length,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	_stdio_ptr,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
	_sig_int_dfl,	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
getw	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_signal,	_sig_segv_dfl,
	_semaphore_wait,	_sig_term_dfl,
	_sig_abrt_dfl,	_sig_write_dfl,
	_sig_alloc_dfl,	lseek,
	_sig_fpe_dfl,	read,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
gmtime	_thread_ptr	
hypot	_err_no_ptr	
ldexp	_err_no_ptr,	
ldiv	_err_no_ptr,	
localtime	_err_no_ptr,	_sig_read_dfl,
	_sig_abrt_dfl,	_sig_segv_dfl,
	_sig_alloc_dfl,	_sig_term_dfl,
	_sig_fpe_dfl,	_sig_write_dfl,
	_sig_free_dfl,	_thread_ptr,
	_sig_ill_dfl,	_tzset_ptr,
	_sig_int_dfl,	sbrk
	_sig_null,	
log	_err_no_ptr,	
log10	_err_no_ptr,	
logf	_err_no_ptr,	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
malloc	_err_no_ptr,	_sig_null,
	_sig_abrt_dfl,	_sig_read_dfl,
	_sig_alloc_dfl,	_sig_segv_dfl,
	_sig_fpe_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl,
	_sig_ill_dfl,	sbrk
	_sig_int_dfl,	
mktime	_err_no_ptr,	_sig_read_dfl,
	_sig_abrt_dfl,	_sig_segv_dfl,
	_sig_alloc_dfl,	_sig_term_dfl,
	_sig_fpe_dfl,	_sig_write_dfl,
	_sig_free_dfl,	_thread_ptr,
	_sig_ill_dfl,	_tzset_ptr,
	_sig_int_dfl,	sbrk
	_sig_null,	
perror	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	_stdio_ptr,
	_map_length,	lseek,
	_semaphore_signal,	write
pow	_err_no_ptr	
powf	_err_no_ptr,	
printf	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	_stdio_ptr,
	_map_length,	lseek,
	_semaphore_signal,	write

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
putc	_err_no_ptr,	_sig_int_dfl,
	_semaphore_signal,	_sig_null,
	_semaphore_wait,	_sig_read_dfl,
	_sig_abrt_dfl,	_sig_segv_dfl,
	_sig_alloc_dfl,	_sig_term_dfl,
	_sig_fpe_dfl,	_sig_write_dfl,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
putchar	_err_no_ptr,	_sig_null,
	_semaphore_signal,	_sig_read_dfl,
	_semaphore_wait,	_sig_segv_dfl,
	_sig_abrt_dfl,	_sig_term_dfl,
	_sig_alloc_dfl,	_sig_write_dfl,
	_sig_fpe_dfl,	_stdio_ptr,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
	_sig_int_dfl,	
puts	_err_no_ptr,	_sig_null,
	_semaphore_signal,	_sig_read_dfl,
	_semaphore_wait,	_sig_segv_dfl,
	_sig_abrt_dfl,	_sig_term_dfl,
	_sig_alloc_dfl,	_sig_write_dfl,
	_sig_fpe_dfl,	_stdio_ptr,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
	_sig_int_dfl,	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
putw	_err_no_ptr,	_sig_int_dfl,
	_semaphore_signal,	_sig_null,
	_semaphore_wait,	_sig_read_dfl,
	_sig_abrt_dfl,	_sig_segv_dfl,
	_sig_alloc_dfl,	_sig_term_dfl,
	_sig_fpe_dfl,	_sig_write_dfl,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	write
raise	_err_no_ptr,	_sig_int_dfl,
	_sig_abrt_dfl,	_sig_null,
	_sig_alloc_dfl,	_sig_read_dfl,
	_sig_fpe_dfl,	_sig_segv_dfl,
	_sig_free_dfl,	_sig_term_dfl,
	_sig_ill_dfl,	_sig_write_dfl,
rand	_thread_ptr	
realloc	_err_no_ptr,	_sig_null,
	_sig_abrt_dfl,	_sig_read_dfl,
	_sig_alloc_dfl,	_sig_segv_dfl,
	_sig_fpe_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl,
	_sig_ill_dfl,	sbrk
	_sig_int_dfl,	
remove	_err_no_ptr,	unlink
rewind	_err_no_ptr,	_semaphore_wait,
	_exit_ptr,	lseek,
	_map_length,	write
	_semaphore_signal,	

continued ☞

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
rmtmp	_err_no_ptr,	_sig_int_dfl,
	_exit_ptr,	_sig_null,
	_map_length,	_sig_read_dfl,
	_semaphore_delete,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	_stdio_ptr,
	_sig_alloc_dfl,	close,
	_sig_fpe_dfl,	lseek,
	_sig_free_dfl,	unlink,
	_sig_ill_dfl,	write
scanf	_err_no_ptr,	_semaphore_wait,
	_semaphore_signal,	_stdio_ptr
setbuf	_err_no_ptr,	_semaphore_wait
	_semaphore_signal,	
setvbuf	_err_no_ptr,	_semaphore_wait
	_semaphore_signal,	
signal	_err_no_ptr,	_sig_int_dfl,
	_sig_abrt_dfl,	_sig_null,
	_sig_alloc_dfl,	_sig_read_dfl,
	_sig_fpe_dfl,	_sig_segv_dfl,
	_sig_free_dfl,	_sig_term_dfl,
	_sig_ill_dfl,	_sig_write_dfl
sin	_errno_ptr	
sinf	_errno_ptr	
sinh	_errno_ptr	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
sqrt	_errno_ptr	
sqrtf	_errno_ptr	
sscanf	_err_no_ptr, _semaphore_signed	_semaphore_wait
strdup	_err_no_ptr, _sig_abrt_dfl, _sig_alloc_dfl, _sig_fpe_dfl, _sig_free_dfl, _sig_ill_dfl,	_sig_int_dfl, _sig_null, _sig_read_dfl, _sig_segv_dfl, _sig_term_dfl, _sig_write_dfl
strerror	_errno_ptr	
strftime	_err_no_ptr, _sig_abrt_dfl, _sig_alloc_dfl, _sig_fpe_dfl, _sig_free_dfl, _sig_ill_dfl, _sig_int_dfl,	_sig_null, _sig_read_dfl, _sig_segv_dfl, _sig_term_dfl, _sig_write_dfl, _tzset_ptr, sbrk
strtod	_errno_ptr	
strtok	_errno_ptr	
strtol	_errno_ptr	
strtoul	_errno_ptr	
svand	_errno_ptr	
tan	_errno_ptr	
tanf	_errno_ptr	
tanh	_errno_ptr	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
tmpfile	_err_no_ptr,	_sig_null,
	_exit_ptr,	_sig_read_dfl,
	_semaphore_init,	_sig_segv_dfl,
	_semaphore_signal,	_sig_term_dfl,
	_semaphore_wait,	_sig_write_dfl,
	_sig_abrt_dfl,	close,
	_sig_alloc_dfl,	isatty,
	_sig_fpe_dfl,	open,
	_sig_free_dfl,	sbrk,
	_sig_ill_dfl,	stat
	_sig_int_dfl,	
tmpnam	_err_no_ptr,	stat
tzset	_err_no_ptr,	_sig_null,
	_sig_abrt_dfl,	_sig_read_dfl,
	_sig_alloc_dfl,	_sig_segv_dfl,
	_sig_fpe_dfl,	_sig_term_dfl,
	_sig_free_dfl,	_sig_write_dfl,
	_sig_ill_dfl,	_tzset_ptr,
	_sig_int_dfl,	sbrk
ungetc	_err_no_ptr,	_sig_int_dfl,
	_semaphore_signal,	_sig_null,
	_semaphore_wait,	_sig_read_dfl,
	_sig_abrt_dfl,	_sig_segv_dfl,
	_sig_alloc_dfl,	_sig_term_dfl,
	_sig_fpe_dfl,	_sig_write_dfl,
	_sig_free_dfl,	sbrk
	_sig_ill_dfl,	

continued ☛

Table A-1 Cross-reference of low-level functions (continued)

This high-level function:	Depends on these low-level functions:	
vfprintf	_err_no_ptr, _exit_ptr, _map_length, _semaphore_signal,	_semaphore_wait, lseek, write
vprintf	_err_no_ptr, _exit_ptr, _map_length, _semaphore_signal,	_semaphore_wait, _stdio_ptr, lseek, write

Index

E

F

G

H

I

N-O

P

R

S