3 C/C++ RUN-TIME LIBRARY

Overview

The C and C++ run-time libraries are collections of functions, macros, and class templates that you can call from your source programs. Many functions are implemented in the ADSP-21xxx family assembly language. C and C++ programs depend on library functions to perform operations that are basic to the C and C++ programming environments. These operations include memory allocations, character and string conversions, and math calculations. Using the library simplifies your software development by providing code for a variety of common needs.

The cc21k compiler provides a broad collection of library functions, including those required by the ANSI standard and additional functions of value for DSP programming supplied by Analog Devices. In addition to the standard C library, this release of the compiler software includes the abridged C++ library, a conforming subset of the standard C++ library. The abridged C++ library includes the embedded C++ and embedded standard template libraries.

This chapter describes the standard C/C++ library functions in the current release of the run-time libraries. Chapter 4, DSP Library for ADSP-2106x Processors, and Chapter 5, DSP Library for ADSP-2116x Processors, describe a number of signal processing, matrix, and statistical functions that assist DSP code development.

For more information on the algorithms on which many of the C library's math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Function*s, Englewood Cliffs, New Jersey: Prentice Hall, 1980. For more information on the C++ library portion of the ANSI/ISO Standard for C++, see Plauger, P. J. (Preface), *The Draft Standard C++ Library,* Englewood Cliffs, New Jersey: Prentice Hall, 1994, (ISBN: 0131170031).

The sections of this chapter present the following information on the compiler:

- "C and C++ Run-Time Libraries Guide" (starting on page 3-3) contains introductory information about the ANSI/ISO Standard C and C++ libraries. It also provides information about the ANSI-standard header files and built-in functions that are included with this release of the cc21k compiler.
- "C Run-Time Library Reference" (starting on page 3-29) contains reference information about the C run-time library functions included with this release of the cc21k compiler.

The C++ library reference information in HTML format is included on the software distribution CD-ROM. To access the reference files from VisualDSP++, see the procedure described in "Related Documents" on page 1-5. Select the *C++ Run-Time Library Reference* from the list of document*s*.

You can also manually access the HTML files using a web browser.

C and C++ Run-Time Libraries Guide

The C and C++ run-time libraries contain routines that you can call from your source program. This section describes how to use the libraries and provides information on the following topics:

For information on the C library's contents, see "C Run-Time Library Reference" (starting on page 3-29). For information on the Abridged C++ library's contents, see "Abridged C++ Library Support" (starting on page 3-21) and on-line Help.

Calling Library Functions

To use a C/C++ library function, call the function by name and give the appropriate arguments. The name and arguments for each function appear on the function's reference page. The reference pages appear in the "C Run-Time Library Reference" section (starting on page 3-29) and in the "C++ Run-Time Library" topic of the on-line Help.

Like other functions you use, library functions should be declared. Declarations are supplied in header files. For more information about the header files see "Working with Library Header Files" (starting on page 3-6).

Function names are C/C++ function names. If you call a C or C++ run-time library function from an assembly program, you must use the assembly version of the function name (prefix an underscore on the name). For more information on the naming conventions, see "C/C++ and Assembly Interface" (starting on page 2-147).

You can use the archiver, `elfar`, described in the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xxx DSPs*, to build library archive files of your own functions.

Linking Library Functions

The C/C++ run-time library is organized as four libraries:

- C run-time library — Comprises all the functions that are defined by the ANSI standard
- C++ run-time library
- DSP run-time library — Contains additional library functions supplied by Analog Devices that provide services commonly required by DSP applications
- I/O library — Supports a subset of the C standard's I/O functionality

Each library has two versions: one set is suitable for running on the ADSP-2106x DSP and the other is suitable for running on the ADSP-2116x DSP. Table 3-1 catalogues the names of each of the libraries and also lists other files that are used while linking a program.

Table 3-1. C and C++ Files and Libraries

Description	ADSP-2106x DSP	ADSP-2116x DSP
C run-time library functions	libc.dlb	libc160.dlb (21160 only) libc161.dlb (21161 only)
Threadsafe C run-time library functions	libcmt.dlb	libc160mt.dlb (21160 only) libc161mt.dlb (21161 only)
C++ run-time library functions	libcpp.dlb	libcpp.dlb
Threadsafe C++ run-time library functions	libcppmt.dlb	libcppmt.dlb
C++ run-time library support functions	libcpprt.dlb	libcpprt.dlb
DSP run-time library functions	libdsp.dlb	libdsp160.dlb
I/O library functions	libio.dlb	libio.dlb
Threadsafe I/O library functions	libiomt.dlb	libiomt.dlb
Start-up file for C programs — calls set-up routines and main	060_hdr.doj (21060 only) 061_hdr.doj (21061 only) 065L_hdr.doj (21065L only)	160_hdr.doj (21160 only) 161_hdr.doj (21161 only)
Start-up file for C++ programs — calls set-up routines and main	060_cpp_hdr.doj (21060 only) 061_cpp_hdr.doj (21061 only) 065L_cpp_hdr.doj (21065L only)	160_cpp_hdr.doj (21160 only) 161_cpp_hdr.doj (21161 only)
Start-up file for multi-threaded C++ applications — calls set-up routines and main	060_cpp_hdr_mt.doj (21060 only) 061_cpp_hdr_mt.doj (21061 only) 065L_cpp_hdr_mt.doj (21065L only)	160_cpp_hdr_mt.doj (21160 only) 161_cpp_hdr_mt.doj (21161 only)

The libraries and start-up files are installed within subdirectories of your VisualDSP++ installation. The files that are used to build applications for the ADSP-2106x architecture are installed within the directory `21k\lib` and those used for the ADSP-2116x architecture are installed within the directory `211xx\lib`.

When you call a run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the library's location is to use the default Linker Description File (`ADSP-<your_target>.ldf`).

If you are not using the default LDF, then either add the appropriate library/libraries to the LDF used for your project, or use the compiler's `-l` switch to specify the library to be added to the link line. For example, the switches `-lc -ldsp` will add `libc.dlb` and `libdsp.dlb` to the list of libraries to be searched by the linker. For more information on the LDF file, see the *VisualDSP++ 2.0 Linker and Utilities Manual for ADSP-21xxx DSPs.*

Working with Library Header Files

When you use a library object (function, macro, or class template) in your program, you include the corresponding header file with the `#include` preprocessor command. The header file for each library function is identified in the Synopsis section of the reference page. Header files contain prototypes. The compiler uses these prototypes to check that each library object is called with the correct arguments.

Standard C Library Header Files

The following C standard header files are supplied with this release of the ADSP-21xxx DSP family compiler. You should use a C standard text to augment the information supplied in this chapter.

Table 3-2 lists the header files that contain ANSI standard run-time environment macros for error handling, standard definitions, limits, and floating-point variables. These do not contain individual functions; they consist of macros and type definitions. See the "Standard C Library Header Files" on page 3-7 for more detailed descriptions.

Table 3-2. ANSI Standard Run-Time Environment Macros

Header File	Description
`errno.h`	Error Handling.
`float.h`	Floating-point implementation parameters.
`limits.h`	Implementation limits.
`stddef.h`	Standard definitions.

Ten C standard header files that contain ANSI standard functions are supplied with the present release of the SHARC compiler. Table 3-3 provides a list of the Standard C Library function header files. See the "Standard C Library Header Files" on page 3-7 for more detailed descriptions.

Table 3-3. ANSI Standard Run-Time Functions

Header File	Description
`assert.h`	Diagnostics.
`ctype.h`	Character handling.

Table 3-3. ANSI Standard Run-Time Functions (Cont'd)

Header File	Description
`locale.h`	Localization.
`math.h`	Basic math functions.
`setjmp.h`	Non-local jumps.
`signal.h`	Signal handling.
`stdarg.h`	Variable arguments.
`stdio.h`	Input/Output.
`stdlib.h`	Standard library.
`string.h`	String handling.

Standard C Library Header File Descriptions

This section provides descriptions of the header files contained in the C library. The header files are listed in alphabetical order.

assert.h

The `assert.h` header file contains the `assert` macro.

ctype.h

The `ctype.h` header file contains functions for character handling, such as `isalpha`, `tolower`, etc.

errno.h

The `errno.h` header file provides access to `errno` and also defines macros for associated error codes.

float.h

The `float.h` header file contains definitions of size and precision values for each C floating point data type. The `FLT_ROUNDS` macro defined in the header file is set to the C run-time environment definition of the rounding mode for `float` variables which is round-toward-nearest. The rounding mode for `long double` variables is truncation.

limits.h

The `limits.h` header file contains definitions of maximum and minimum values for each C data type other than floating-point.

locale.h

The `locale.h` header file contains definitions for expressing numeric, monetary, time, and other data.

math.h

The `math.h` header file includes trigonometric, power, logarithmic, exponential, and other miscellaneous functions. The library contains the functions specified by the C standard along with implementations for `float`.

This header file also provides prototypes for a number of additional math functions provided by Analog Devices, such as `favg`, `fmax`, `fclip`, and `copysign`. Refer to Chapter 4, DSP Library for ADSP-2106x Processors, and Chapter 5, DSP Library for ADSP-2116x Processors, for more information about these additional functions.

The `math.h` header file contains prototypes for single (32-bit), double (64-bit), and faster single precision routines. For every `double` mathematical function, there is a corresponding `float` function. For example, the 32-bit version of `sin` is `sinf`.

If the compiler is treating `double` as 32 bits, the header file defines non-suffixed names (e.g. `sin`) as the 32-bit version (e.g. `sinf`). This lets you use the un-suffixed names with arguments of type `double`, regardless of whether doubles are 32- or 64-bits. The `float` functions offer significant performance improvement.

The `math.h` header file also defines the macro `HUGE_VAL`. `HUGE_VAL` evaluates to the maximum positive value that the type `double` can support.

The macros `EDOM` and `ERANGE`, defined in `errno.h`, are used by `math.h` functions to indicate domain and range errors.

A domain error occurs when an input argument is outside the domain of the function. The section "C Run-Time Library Reference" (starting on page 3-29) lists the specific cases that cause `errno` to be set to `EDOM`, and the associated return values.

A range error occurs when the result of a function cannot be represented in the return type. If the result overflows, the function returns the value `HUGE_VAL` with the appropriate sign. If the result underflows, the function returns a zero without indicating a range error.

setjmp.h

The `setjmp.h` header file contains `setjmp` and `longjmp` for non-local jumps.

signal.h

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several ADSP-21xxx family extensions, such as `interrupt()` and `clear_interrupt()`.

The signal handling functions process conditions (hardware signals) that can occur during program execution. They determine the way that your C program responds to these signals. The functions are designed to process such signals as external interrupts and timer interrupts.

Interrupt Dispatchers. Four interrupt dispatchers let you disable interrupts and modify the processor's `MODE1` register from an interrupt handler. The interrupt dispatchers are described below.

For the normal interrupt dispatcher, use the `interrupt()` or `signal()` functions. (The interrupt dispatcher normally uses self-modifying code. If this is not suitable for your application, then you should use the `interruptcbnsm` or `signalcbnsm` functions instead.)

This dispatcher provides the following services:

- Saves all scratch registers and the loop stack. Do loop and interrupt nesting is allowed because data is pushed onto the stack. Requires approximately 125 cycles for interrupt overhead.
- Preserves MODE1 register writes in an interrupt handler after the interrupt is serviced.
- Freezes the cache. An example, which is found in `21k\src\crt_src\060_hdr.asm`, is shown below:

```
___z3_int_determiner:
     DM(I7,M7)=I13;         /* Save I13 (scratch reg)   */
     I13=PM(5,I15);         /* get disp to jump to      */
     JUMP (M13, I13) (DB);  /* Jump to dispatcher       */
     BIT SET MODE2 0x80000; /* Freeze cache             */
     I13=PM(2,I15);         /* rd handler addr (base+2) */
```

The circular buffer dispatcher is supplied for users whose code may generate interrupts during the execution of circular buffering code and whose interrupt handlers may be affected by the fact that the L registers are non-zero. For the circular buffer interrupt dispatcher, use the `interruptcb()` or `signalcb()` functions. (The interrupt dispatcher normally uses self-modifying code. If this is not suitable for your application, then you should use the `interruptcbnsm` or `signalcbnsm` functions instead.)

This dispatcher is the same as the normal interrupt dispatcher, but it provides the following additional services:

- Before calling the handler, the registers `L0-L5` and `L8-L15` are saved and set to zero.
- After executing the handler, the L registers are restored to their original values.
- This dispatcher requires an additional 54 cycles to save, zero, and restore the L registers.

For the fast interrupt dispatcher, use the `interruptf()` or `signalf()` functions. (The interrupt dispatcher normally uses self-modifying code. If this is not suitable for your application, then you should use the `interruptcbnsm` or `signalcbnsm` functions instead.)

This dispatcher provides the following services:

- Does not save the loop stack; DO loop handling is restricted to six levels (specified in hardware). If the interrupt service routine (ISR) uses one level of nesting, your code cannot exceed five levels.
- Preserves `MODE1` register writes in an interrupt handler after the interrupt is serviced.
- Interrupt nesting is not restricted (20 levels available). Does not send the interrupt number type to the ISR as a parameter. Requires approximately 60 cycles for interrupt overhead.

For the super interrupt dispatcher, use the `interrupts()` or `signals()` functions. (The interrupt dispatcher normally uses self-modifying code. If this is not suitable for your application, then you should use the `interruptcbnsm` or `signalcbnsm` functions instead.)

This dispatcher provides the following services:

- Does not save the loop stack, therefore do loop handling is restricted to six levels (specified in hardware). Interrupt nesting is disabled. This dispatcher does not send the interrupt number type to the ISR as a parameter.
- Uses the secondary register set. This dispatcher requires approximately 30 cycles for interrupt overhead.

the choice of a non self-modifying function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

stdarg.h

The `stdarg.h` header file contains definitions needed for functions that accept a variable number of arguments. Callers of such functions must include a prototype.

stddef.h

The `stddef.h` header file contains a few common definitions useful for portable programs, such as `size_t`.

stdio.h

The `stdio.h` header file contains a subset of the C standard's I/O functionality. Always include the header file in your source if you use any of its facilities because the header file contains dual support for type `double` — support for when it is 32 bits and support for when it is 64 bits. Failure to include the header file results in a linker failure as the compiler must see a correct function prototype in order to generate the correct calling sequence.

The following facilities are not available in this release:

- Stream positioning functions `fgetpos`, `fseek`, `fsetpos`, `ftell`, and `rewind`
- File handling functions `remove`, `rename`, `tmpfile`, and `tmpnam`
- `printf` and `scanf` functions do not support values of type `long long`
- `printf` and `scanf` functions do not support values of type `long double` when type `double` is the same size as type `float`

The functions supported by the `stdio.h` header use a simple interface with a host environment, which may be a simulator, debugger, or an external downloader (e.g., ELFDLK for SHARC processor boards). The intent is to place minimal demands on the host environment while still providing as much functionality to the DSP as possible.

All I/O operations are channeled through the C function `_primIO`. The assembly label has two underscores, `__primIO`. `__primIO` accepts no arguments. Instead, it examines the I/O control block at the label `_PrimIOCB`. Without external intervention by a host environment, the `__primIO` routine simply returns, which indicates failure of the request. Two schemes for host interception of I/O requests are provided.

The first scheme is to modify control flow into and out of the _primIO routine. Typically, this would be achieved by a break point mechanism available to a debugger/simulator. Upon entry to _primIO, the data for the request will reside in a control block at the label _PrimIOCB. If this scheme is used, the host should arrange to intercept control when it enters the _primIO routine, and, after servicing the request, return control to the calling routine.

The second scheme involves communicating with the DSP process through a pair of simple semaphores. This scheme is most suitable for an externally-hosted development board. Under this scheme, the host system should clear the data word whose label is __lone_SHARC; this will cause _primIO to assume that a host environment is present and able to communicate with the DSP process. If _primIO sees that __lone_SHARC is cleared, then upon entry (i.e., when an I/O request is made) it will set a non-zero value into the word labeled __Godot. _primIO will then busy-wait until this word is reset to zero by the host. The non-zero value of __Godot raised by _primIO is the address of the I/O control block.

Data Packing For Primitive I/O

The DSP implementation is based on a word-addressable machine, with each word comprising a fixed number of 8-bit bytes. All READ and WRITE requests specify a move of some number of 8-bit bytes, that is, the relevant fields count 8-bit bytes, not words. Packing is always little endian, the first byte of a file read or written is the low-order byte of the first word transferred.

Data packing is set to four bytes per word for the SHARC. Data packing can be changed on the DSP side to accommodate other DSP architectures by modifying the constant BITS_PER_WORD, defined in wordsize.h.

Note that the file name provided in an OPEN request uses the DSP's "native" string format, normally one byte per word. Data packing applies only to READ and WRITE requests.

Data Structure for Primitive I/O

The I/O control block is declared in `_primio.h`, as follows:

```
typedef struct
{
    enum
    {
        PRIM_OPEN = 100,
        PRIM_READ,
        PRIM_WRITE,
        PRIM_CLOSE
    } op;

    int    fileID;
    int    flags;
    unsigned char *buf;   /* data buffer, or file name */
    int    nDesired;      /* number of characters to read */
                          /* or write */
    int    nCompleted;    /* number of characters actually */
                          /* read or written */
    void  *more;          /* for future use */
}
PrimIOCB_T;
```

The first field, `op`, identifies which of the four currently supported operations is being requested.

The file ID for an open file is a non-negative integer assigned by the debugger or other “host” mechanism. The fileID values 0, 1, and 2 are pre-assigned to `stdin`, `stdout`, and `stderr`, respectively. No open request is required for these file IDs.

The `flags` field is a bit field containing other information for special requests. Meaningful bit values for an `OPEN` operation are:

```
M_OPENR = 0x0001    /* open for reading */
M_OPENW = 0x0002    /* open for writing */
M_OPENA = 0x0004    /* open for append */
M_TRUNCATE = 0x0008 /* truncate to zero length if file exists */
M_CREATE = 0x0010   /* create the file if necessary */
M_BINARY = 0x0020   /* binary file (vs. text file) */
```

For a READ operation, the low-order four bits of the flag value contain the number of bytes packed into each word of the read buffer, and the rest of the value are reserved for future use.

For a WRITE operation, the low-order four bits of the flag value contain the number of bytes packed into each word of the write buffer, and the rest of the value form a bit field, for which only the following bit is currently defined:

```
M_ALIGN_BUFFER = 0x10
```

If this bit is set for a WRITE request, the WRITE operation is expected to be aligned on a DSP word boundary by writing padding NULs to the file before the buffer contents are transferred.

The flags field is currently unused for a CLOSE request.

The buf field contains a pointer to the file name for an open request, or a pointer to the data buffer for a read or write request.

For a read or write request, nDesired is the number of bytes the program is requesting to transfer; _primIO is expected to set nCompleted to the number of bytes actually transferred.

The more field is reserved for future use, and currently is always set to NULL before calling _primIO.

stdlib.h

The stdlib.h header file offers general utilities specified by the C standard. These include some integer math functions, such as abs, div, and rand; general string-to-numeric conversions; memory allocation functions, such as malloc and free; and termination functions, such as exit. This library also contains miscellaneous functions such as bsearch and qsort.

This header file also provides prototypes for a number of additional integer math functions provided by Analog, such as avg, max, and clip.

Table 3-4 provides a summary of the additional library functions defined by header file.

Table 3-4. Standard Library - Additional Functions

Description	Prototype
Average	`int avg (int a, int b);` `long lavg (long a, long b);`
Clip	`int clip (int a, int b);` `long lclip (long a, long b);`
Maximum	`int max (int a, int b);` `long lmax (long a, long b);`
Minimum	`int min (int a, int b);` `long lmin (long a, long b);`

Multiple heaps for dynamic memory allocation	`int heap_init(int heap_index);`
	`int heap_lookup(int user_id);`
	`void *heap_calloc(int heap_index, size_t nelem, size_t size);`
	`void heap_free(int heap_index, void *ptr);`
	`void *heap_malloc(int heap_index, size_t size);`
	`void *heap_realloc(int heap_index, void *ptr, size_t size);`
	`int heap_switch(int heap_index);`

Some functions exist as both integer and floating point. The floating-point functions typically have an `f` prefix. Make sure you use the correct type.

A number of functions, including `abs`, `avg`, `max`, `min`, and `clip`, are implemented via intrinsics (provided the header file has been `#include`'d) that map to single-cycle machine instructions.

If the header file is not included, the library implementation is used instead — at a considerable loss in efficiency.

string.h

The `string.h` header file contains string handling functions, including `strcpy` and `memcpy`.

Using the Compiler's Built-In C library Functions

The C compiler's intrinsic (built-in) functions are functions that the compiler immediately recognizes and replaces with in-line assembly code instead of a function call. For example, the absolute value function, `abs()`, is recognized by the compiler, which subsequently replaces a call to the C run-time library version with an in-line version. The cc21k compiler contains a number of intrinsic built-in functions for efficient access to various features of the hardware.

Built-in functions are recognized for cases where the name begins with the string `__builtin`, and the declared prototype of the function matches the prototype that the compiler expects. Built-in functions are declared in system header files. Include the appropriate header file in your program to use these functions. The normal action of the appropriate include file is to `#define` the normal name as mapping to the built-in form.

Typically, in-line assembly code is faster than an average library routine, and it does not incur the calling overhead.

The routines in Table 3-5 are built-in C library functions for the cc21k compiler:

Table 3-5. Compiler Built-in Functions

abs	avg	clip
copysign[1]	copysignf	fabs[1]
fabsf	favg[1]	favgf
fclip[1]	fclipf	fmax[1]
fmaxf	fmin[1]	fminf
labs	lavg	lclip
lmax	lmin	max
min		

[1] These functions will only be compiled as a built-in function if double is the same size as float.

If you want to use the C run-time library functions of the same name, compile with the `-no-builtin` (no built-in functions) compiler switch.

For a certain category of library function, the compiler relaxes the normal rule whereby pointers that are passed as arguments must address Data Memory (DM). For functions in this category, any argument that is a pointer may also address Program Memory (PM). When the compiler recognizes that certain arguments reference PM, it generates a call to an appropriate version of the function in the run-time library.

Table 3-6 contains a list of library functions that may be called with pointers to Program Memory. Note that this facility is only available provided that the compiler switch `-no-builtin` has not been specified.

Table 3-6. Dual Memory Capable Functions

atof	atoi	atol
memchr	memcmp	memcpy
memmove	memset	modf
modff	setlocale	strcat
strchr	strcmp	strcoll
strcpy	strcspn	strlen
strncat	strncmp	strncpy
strpbrk	strrchr	strspn
strstr	strtod	strtok
strtol	strtoul	strxfrm

Abridged C++ Library Support

When in C++ mode, the cc21k compiler can call a large number of functions from the Abridged Library, a conforming subset of C++ library.

The Abridged Library has two major components: Embedded C++ Library (EC++) and Embedded Standard Template Library (ESTL). The Embedded C++ Library is a conforming implementation of the Embedded C++ Library as specified by the Embedded C++ Technical Committee.

This section lists and briefly describes the following components of the Abridged Library:

- "Embedded C++ Library Header Files" on page 3-22
- "C++ Header Files for C Library Facilities" on page 3-25
- "Embedded Standard Template Library Header Files" on page 3-26

For more information on the Abridged Library, see online Help.

Embedded C++ Library Header Files

complex

The `complex` header file defines a template that supports the `double_complex` and `float_complex` classes and a set of arithmetic operators.

exception

The `exception` header file defines the `exception` and `bad_exception` classes and several functions for exception handling.

fract

The `fract` header file defines the `fract` data type, which supports fractional arithmetic, assignment, and type-conversion operations. The header file is fully described under "C++ Fractional Type Support" on page 2-84. An example that demonstrates its use appears under "C++ Programming Examples" on page 2-162.

fstream

The `fstream` header file defines the `filebuf`, `ifstream`, and `ofstream` classes for external file manipulations.

iomanip

The `iomanip` header file declares several iostream manipulators. Each manipulator accepts a single argument.

ios

The `ios` header file defines several classes and functions for basic iostream manipulations. Note that most of the iostream header files include `ios.h`.

iosfwd

The `iosfwd` header file declares forward references to various iostream template classes defined in other standard header files.

iostream

The `iostream` header file declares most of the iostream objects used for the standard stream manipulations.

istream

The `istream` header file defines the `istream` class for iostream extractions. Note that most of the iostream header files include `istream.h`.

new

The `new` header file declares several classes and functions for memory allocations and decollations.

ostream

The `ostream` header file defines the `ostream` class for iostream insertions.

sstream

The `sstream` header file defines the `stringbuf`, `istringstream`, and `ostringstream` classes for various `string` object manipulations.

stdexcept

The `stdexcept` header file defines a variety of classes for exception reporting.

streambuf

The `streambuf` header file defines the `streambuf` classes for basic operations of the iostream classes. Note that most of the iostream header files include `streambuf.h`.

string

The `string` header file defines the `string` template and various supporting classes and functions for string manipulations.

Objects of the `string` type should not be confused with the NULL-terminated C strings.

strstream

The `strstream` header file defines the `strstreambuf`, `istrstream`, and `ostream` classes for iostream manipulations on allocated, extended, and freed character sequences.

C++ Header Files for C Library Facilities

For each C standard library header there is a corresponding standard C++ header. If the name of a C standard library header file is `foo.h`, then the name of the equivalent C++ header file will be `cfoo`. For example, the C++ header file `<cstdio>` provides the same facilities as the C header file `<stdio.h>`. Table 3-7 lists the C++ header files that provide access to the C library facilities.

Normally, the C standard headers files may be used to define names in the C++ global namespace while the equivalent C++ header files define names in the std namespace. However, the std namespace is not supported in this release of the compiler, and the effect of including one of the C++ header files listed in Table 3-7 is the same as including the equivalent C standard library header file.

Table 3-7. C++ Header Files for C Library Facilities

Header	Description
`<cassert>`	Enforces assertions during function executions
`<cctype>`	Classifies characters
`<cerrno>`	Tests error codes reported by library functions
`<cfloat>`	Tests floating-point type properties
`<climits>`	Tests integer type properties
`<clocale>`	Adapts to different cultural conventions
`<cmath>`	Provides common mathematical operations
`<csetjmp>`	Executes non-local goto statements
`<csignal>`	Controls various exceptional conditions
`<cstdarg>`	Accesses a various number of arguments

Table 3-7. C++ Header Files for C Library Facilities (Cont'd)

Header	Description
<cstddef>	Defines several useful data types and macros
<cstdio>	Performs input and output
<cstdlib>	Performs a variety of operations
<cstring>	Manipulates several kinds of strings

Embedded Standard Template Library Header Files

Templates and the associated header files are not part of the Embedded C++ standard, but they are supported by the cc21k compiler in C++ mode. The fifteen Embedded Standard Template Library header files are:

<algorithm>

The <algorithm> header file defines numerous common operations on sequences.

<deque>

The <deque> header file defines a deque template container.

<functional>

The <functional> header file defines numerous function objects.

<hash_map>

The <hash_map> header file defines two hashed map template containers.

<hash_set>

The <hash_set> header file defines two hashed set template containers.

<iterator>

The <iterator> header file defines common iterators and operations on iterators.

<list>

The <list> header file defines a list template container.

<map>

The <map> header file defines two map template containers.

<memory>

The <memory> header file defines facilities for managing memory.

<numeric>

The <numeric> header file defines several numeric operations on sequences.

<queue>

The <queue> header file defines two queue template container adapters.

<set>

The <set> header file defines two set template containers.

<stack>

The <stack> header file defines a stack template container adapter.

<utility>

The <utility> header file defines an assortment of utility templates.

<vector>

The `<vector>` header file defines a vector template container.

The Embedded C++ library also includes several header files for compatibility with traditional C++ libraries:

fstream.h

The `fstream.h` header file defines several iostreams template classes that manipulate external files.

iomanip.h

The `iomanip.h` header file declares several iostreams manipulators that take a single argument.

iostream.h

The `iostream.h` header file declares the iostreams objects that manipulate the standard streams.

new.h

The `new.h` header file declares several functions that allocate and free storage.

C Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C programs. This section lists the functions in alphabetical order. Note the following items that apply to all the functions in the library.

Notation Conventions. An interval of numbers is indicated by the minimum and maximum, separated by a comma, and enclosed in two square brackets, two parentheses, or one of each. A square bracket indicates that the endpoint is included in the set of numbers; a parenthesis indicates that the endpoint is not included.

Function Benchmarks and Specifications. All functions have been timed from setup, to invocation, to results storage of returned value. This includes all register storing, parameter passing, etc. Most functions execute slightly faster if you pass constants as arguments instead of variables.

Restrictions. When polymorphic functions are used and the function returns a pointer to program memory, cast the output of the function to pm. For example:

```
(char pm *)
```

Reference Format. Each function in the library has a reference page. These pages follow the following format:

Name and Purpose of the function

Synopsis—Required header file and functional prototype

Description—Function specification

Error Conditions—How the function indicates an error

Example—Typical function usage

See Also—Related functions

abort

abnormal program end

Synopsis

```
#include <stdlib.h>
void abort (void);
```

Description

The function `abort` causes an abnormal program termination by raising the `SIGABRT` exception. The last action of the default abort handler passes control to the label `___lib_prog_term` which is defined in the run-time startup file.

Error Conditions

The `abort` function does not return.

Example

```
#include <stdlib.h>
extern int errors;

if (errors)    /* terminate program if */
    abort();   /* errors are present   */
```

See Also

atexit, exit

abs

absolute value

Synopsis

```
#include <stdlib.h>
int abs (int j);
```

Description

The `abs` function returns the absolute value of its integer argument.

Note: `abs(INT_MIN)` returns `INT_MIN`.

Error Conditions

The `abs` function does not return an error condition.

Example

```
#include <stdlib.h>
int i;

i  = abs (-5);     /* i == 5 */
```

See Also

fabs, fabsf, labs

acos, acosf

arc cosine

Synopsis

```
#include <math.h>
double acos (double x);
float acosf  (float x);
```

Description

The `acos` and `acosf` functions return the arc cosine of `x`. The input must be in the range [-1, 1]. The output, in radians, is in the range [0, π].

The `acos` and `acosf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `acos` and `acosf` functions indicate a domain error (set `errno` to `EDOM`) and returns a zero if the input is not in the range [–1, 1].

Example

```
#include <math.h>
double x;
float y;

y  = acos (0.0);    /* y = π/2 */
x  = acosf (0.0);   /* x = π/2 */
```

See Also

cos, cosf

asin, asinf

arc sine

Synopsis

```
#include <math.h>
double asin (double x);
float asinf (float x);
```

Description

The `asin` and `asinf` functions return the arc sine of the first argument. The input must be in the range [1, 1]. The output, in radians, is in the range

$$[-\frac{\pi}{2}, \frac{\pi}{2}]$$

The `asin` and `asinf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `asin` and `asinf` functions indicate a domain error (set `errno` to `EDOM`) and return a zero if the input is not in the range [-1, 1].

Example

```
#include <math.h>
double y;
float x;

y = asin (1.0);     /* y = π/2  */
x = asinf (1.0);    /* x = π/2  */
```

See Also

sin, sinf

atan, atanf

arc tangent

Synopsis

```
#include <math.h>
double atan (double x);
float atanf (float x);
```

Description

The `atan` and `atanf` functions return the arc tangent of the first argument.

The output, in radians, is in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$

The `atan` and `atanf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `atan` and `atanf` functions do not return error conditions.

Example

```
#include <math.h>
double y;
float x;

y = atan (0.0);     /* y = 0.0 */
x = atanf (0.0);    /* x = 0.0 */
```

See Also

atan2, atan2f, tan, tanf

atan2, atan2f

arc tangent of quotient

Synopsis

```
#include <math.h>
double atan2 (double x, double y);
float atan2f (float x, float y);
```

Description

The `atan2` and `atan2f` functions compute the arc tangent of the input value `x` divided by input value `y`. The output, in radians, is in the range [-π, π].

The `atan2` and `atan2f` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `atan2` and `atan2f` functions return a zero and set `errno` to `EDOM` if `x=0` and `y <> 0`.

Example

```
#include <math.h>
double a;
float b;

a = atan2 (0.0, 0.5);    /* the error condition: a = 0.0 */
b = atan2f (1.0, 0.0);   /*  b = π/2 */
```

See Also

atan, atanf, tan, tanf

atexit

register a function to call at program termination

Synopsis

```
#include <stdlib.h>
int atexit (void (*func)(void));
```

Description

The `atexit` function registers a function to be called at program termination. Functions are called once for each time they are registered, in the reverse order of registration. Up to 32 functions can be registered using `atexit`.

Error Conditions

The `atexit` function returns a nonzero value if the function cannot be registered.

Example

```
#include <stdlib.h>
extern void goodbye(void);

if (atexit(goodbye))
   exit(1);
```

See Also

abort, exit

atof

convert string to a double

Synopsis

```
#include <stdlib.h>
double atof (const char *nptr);
```

Description

The `atof` function converts a character string to a double value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign). Conversion terminates at the first non-digit (exceptions are ".", "`e`", "`E`", and exponents, including the sign).

There is no way to determine if a zero is a valid result or an indicator of an invalid string.

This function requires the use of the compiler's `-double-size-64` switch.

Error Conditions

The `atof` function returns a zero if no conversion can be made.

Example

```
#include <stdlib.h>
double x;

x = atof ("5.5");   /* x == 5.5 */
```

See Also

atoi, atol, strtol, strtoul

atoi

convert string to integer

Synopsis

```
#include <stdlib.h>
int atoi (const char *nptr);
```

Description

The `atoi` function converts a character string to an integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

Error Conditions

The `atoi` function returns a zero if no conversion can be made.

Example

```
#include <stdlib.h>
int i;

i = atoi ("5");   /* i == 5 */
```

See Also

atof, atol, strtod, strtol, strtoul

atol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long atol (const char *nptr);
```

Description

The `atol` function converts a character string to a long integer value. The character string to be converted is pointed to by the input pointer, `nptr`. The function clears any leading characters for which `isspace` would return true. Conversion begins at the first digit (with an optional preceding sign) and terminates at the first non-digit.

There is no way to determine if a zero is a valid result or an indicator of an invalid string.

Error Conditions

The `atol` function returns a zero if no conversion can be made.

Example

```
#include <stdlib.h>
long int i;

i = atol ("5");   /* i == 5 */
```

See Also

atoi, atof, strtod, strtol, strtoul

avg

returns mean of two values

Synopsis

```
#include <stdlib.h>
int avg (int x, int y);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `avg` function adds two arguments and divides the result by two. The `avg` function is a built-in function which is implemented with an `Rn=(Rx+Ry)/2` instruction.

Error Conditions

The `avg` function does not return an error code.

Example

```
#include <stdlib.h>
int i;

i = avg (10, 8);   /* returns 9 */
```

See Also

favg, favgf, lavg

bsearch

perform binary search in a sorted array

Synopsis

```
#include <stdlib.h>
void *bsearch (const void *key, const void *base,
 size_t nelem, size_t size,
 int (*compare)(const void *, const void *));
```

Description

The `bsearch` function executes a binary search operation on a pre-sorted array, where:

- `key` is a pointer to the element to search for.
- `base` points to the start of the array.
- `nelem` is the number of elements in the array. The type `size_t` is defined in `stdlib.h` as

  ```
  typedef long unsigned int size_t;
  ```

- `size` is the size of each element of the array.
- `*compare` points to the function used to compare two elements. It takes as parameters a pointer to the key and a pointer to an array element. The function should return a value less than, equal to, or greater than zero according to whether the first parameter is less than, equal to, or greater than the second.

The `bsearch` function returns a pointer to the first occurrence of `key` in the array.

Error Conditions

The `bsearch` function returns NULL if the key is not found in the array.

Example

```
#include <stdlib.h>
char *answer;
char base[50][3];

answer = bsearch ("g", base, 50, 3, strcmp);
```

See Also

qsort

calloc

allocate and initialize memory

Synopsis

```
#include <stdlib.h>
void *calloc (size_t nmemb, size_t size);
```

Description

The `calloc` function dynamically allocates a range of memory and initializes all locations to zero. The number of elements (the first argument) multiplied by the size of each element (the second argument) is the total memory allocated. The memory may be deallocated with the `free` function.

The object is allocated from the current heap, which is the default heap unless `heap_switch` has been called to change the current heap to an alternate heap.

The type `size_t` is defined in `stdlib.h` as

```
typedef long unsigned int size_t;
```

Error Conditions

The `calloc` function returns the NULL pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *) calloc (10, sizeof (int));
/* ptr points to a zeroed array of length 10 */
```

See Also

free, heap_calloc, heap_free, heap_init, heap_lookup, heap_malloc, heap_realloc, heap_switch, malloc, realloc

ceil, ceilf

ceiling

Synopsis

```
#include <math.h>
double ceil (double x);
float ceilf (float x);
```

Description

The `ceil` function returns the smallest integral value, expressed as `double`, that is not less than its argument. The `ceilf` function returns the smallest integral value, expressed as `float`, that is not less than its input.

Error Conditions

The `ceil` and `ceilf` functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;

y = ceil (1.05);      /* y = 2.0  */
x = ceilf (-1.05);    /* y = -1.0 */
```

See Also

floor, floorf

clear_interrupt

clear a pending signal

Synopsis

```
#include <signal.h>
int clear_interrupt (int sig);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `clear_interrupt` function clears the signal `sig` in the `IRPTL` register. For the ADSP-2106x processors, the `sig` argument must be one of the processor signals shown below in Table 3-8, and for the ADSP-2116x processors, the `sig` argument must be one of the processor signals in Table 3-9. This function does not work for interrupts that set any status bits in the `STKY` register, such as floating-point overflow.

Table 3-8. ADSP-2106x Signals

Sig Value	Definition
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_VIRPTI	Vector Interrupt
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_SPR0I	DMA Channel 0 - SPORT0 Receive
SIG_SPR1I	DMA Channel 1 - SPORT1 Receive (or Link Buffer 0)

Table 3-8. ADSP-2106x Signals (Cont'd)

Sig Value	Definition
SIG_SPT0I	DMA Channel 2 - SPORT0 Transmit
SIG_SPT1I	DMA Channel 3 - SPORT1 Transmit (or Link Buffer 1)
[1]SIG_LP2I	DMA Channel 4 - Link Buffer 2
[1]SIG_LP3I	DMA Channel 5 - Link Buffer 3
SIG_EP0I	DMA Channel 6 - Ext. Port Buffer 0 (or Link Buffer 4)
SIG_EP1I	DMA Channel 7 - Ext. Port Buffer 1 (or Link Buffer 5)
[1]SIG_EP2I	DMA Channel 8 - Ext. Port Buffer 2
[1]SIG_EP3I	DMA Channel 9 - Ext. Port Buffer 3
[1]SIG_LSRQ	Link port service request
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed point overflow
SIG_FLTO	Floating point overflow exception
SIG_FLTU	Floating point underflow exception
SIG_FLTI	Floating point invalid exception
SIG_USR0	User software interrupt 0
SIG_USR1	User software interrupt 1

Table 3-8. ADSP-2106x Signals (Cont'd)

Sig Value	Definition
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3

[1] Signal is not present on the ADSP-21061 and ADSP-21065L processors.

Table 3-9. ADSP-2116x Signals

Sig Value	Definition
SIG_IICDI	Illegal input condition detected
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_VIRPTI	Vector Interrupt
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_SPR0I	DMA Channel 0 - SPORT0 Receive
SIG_SPR1I	DMA Channel 1 - SPORT1 Receive
SIG_SPT0I	DMA Channel 2 - SPORT0 Transmit
SIG_SPT1I	DMA Channel 3 - SPORT1 Transmit
SIG_LP0I	DMA Channel 4 - Link Buffer 0

Table 3-9. ADSP-2116x Signals (Cont'd)

Sig Value	Definition
SIG_LP1I	DMA Channel 5 - Link Buffer 1
SIG_LP2I	DMA Channel 6 - Link Buffer 2
SIG_LP3I	DMA Channel 7 - Link Buffer 3
SIG_LP4I	DMA Channel 8 - Link Buffer 4
SIG_LP5I	DMA Channel 9 - Link Buffer 5
SIG_EP0I	DMA Channel 10 - Ext. Port Buffer 0
SIG_EP1I	DMA Channel 11 - Ext. Port Buffer 1
SIG_EP2I	DMA Channel 12 - Ext. Port Buffer 2
SIG_EP3I	DMA Channel 13 - Ext. Port Buffer 3
SIG_LSRQ	Link port service request
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed point overflow
SIG_FLTO	Floating point overflow exception
SIG_FLTU	Floating point underflow exception
SIG_FLTI	Floating point invalid exception
SIG_USR0	User software interrupt 0
SIG_USR1	User software interrupt 1

Table 3-9. ADSP-2116x Signals (Cont'd)

Sig Value	Definition
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3

Error Conditions

The `clear_interrupt` function returns a 1 if the interrupt was pending; otherwise 0 is returned.

Example

```
#include <signal.h>

clear_interrupt (SIG_IRQ2);
/* clear the interrupt 2 latch */
```

See Also

interrupt, raise, signal

clip

clip x by y

Synopsis

```
#include <stdlib.h>
int clip (int value1, int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `clip` function returns its first argument if it is less than the absolute value of its second argument, otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative. The `clip` function is a built-in function which is implemented with an `Rn=CLIP Rx BY Ry` instruction.

Error Conditions

The `clip` function does not return an error code.

Example

```
#include <stdlib.h>
int i;

i = clip (10, 8);    /* returns 8  */
i = clip (8, 10);    /* returns 8  */
i = clip (-10, 8);   /* returns -8 */
```

See Also

fclip, fclipf

cos, cosf

cosine

Synopsis

```
#include <math.h>
double cos (double x);
float cosf (float x);
```

Description

The `cos` and `cosf` functions return the cosine of the first argument. The input is interpreted as radians; the output is in the range [-1, 1].

The `cos` and `cosf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range. Although the `cos` and `cosf` functions accept input over the entire floating-point range, the accuracy of the result decreases significantly for an input greater than $\pi^{12}/2$.

Error Conditions

The `cos` and `cosf` functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;

y = cos (3.14159);     /* y = -1.0*/
x = cosf (3.14159);    /* x = -1.0*/
```

See Also

acos, acosf, sin, sinf

cosh, coshf

hyperbolic cosine

Synopsis

```
#include <math.h>
double cosh (double x);
float coshf (float x);
```

Description

The `cosh` and `coshf` functions return the hyperbolic cosine of their argument.

The `cosh` and `coshf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `cosh` and `coshf` functions return `HUGE_VAL` and set `errno` to `ERANGE` if the input exceeds 2^{12}.

Example

```
#include <math.h>
double x, y;
float v, w;

y = cosh (x);
v = coshf (w);
```

See Also

sinh, sinhf

div

division

Synopsis

```
#include <stdlib.h>
div_t div (int numer, int denom);
```

Description

The `div` function divides `numer` by `denom`, both of type `int`, and returns a structure of type `div_t`. The type `div_t` is defined as

```
typedef struct {
   int quot;
   int rem;
} div_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if `result` is of type `div_t`,

```
result.quot * denom + result.rem == numer
```

Error Conditions

If `denom` is zero, the behavior of the `div` function is undefined.

Example

```
#include <stdlib.h>
div_t result;

result = div (5, 2);   /* result.quot = 2, result.rem = 1 */
```

See Also

ldiv, fmod, fmodf, modf, modff

exit

normal program termination

Synopsis

```
#include <stdlib.h>
void exit (int status);
```

Description

The `exit` function causes normal program termination. The functions registered by the `atexit` function are called in reverse order of their registration and the microprocessor is put into the `IDLE` state. The `status` argument is stored in register `R0`, and control is passed to the label `___lib_prog_term`, which is defined in the run-time startup file.

Error Conditions

The `exit` function does not return an error condition.

Example

```
#include <stdlib.h>

exit (EXIT_SUCCESS);
```

See Also

abort, atexit

exp, expf

exponential

Synopsis

```
#include <math.h>
double exp (double x);
float expf (float x);
```

Description

The `exp` and `expf` functions compute the exponential value `e` to the power of its argument.

Error Conditions

For underflow errors the `exp` and `expf` functions return zero.

Example

```
#include <math.h>
double y;
float x;

y = exp (1.0);    /* y = 2.71828... */
x = expf (1.0);   /* x = 2.71828... */
```

See Also

log, logf, pow, powf

fabs, fabsf

absolute value

Synopsis

```
#include <math.h>
double fabs (double x);
float fabsf (float x);
```

Description

The `fabs` and `fabsf` functions return the absolute value of the argument.

Error Conditions

The `fabs` and `fabsf` functions do not return error conditions.

Example

```
#include <math.h>
double y;
float x;

y = fabs (-2.3);      /* y = 2.3 */
y = fabs (2.3);       /* y = 2.3 */
x = fabsf (-5.1);     /* x = 5.1 */
```

See Also

abs, labs

floor, floorf

floor

Synopsis

```
#include <math.h>
double floor (double x);
float floorf (float x);
```

Description

The `floor` and `floorf` functions return the largest integral value that is not greater than their argument.

Error Conditions

The `floor` and `floorf` functions do not return error conditions.

Example

```
#include <math.h>
double y;
float z;

y = floor (1.25);      /* y = 1.0*/
y = floor (-1.25);     /* y = -2.0*/
z = floorf (10.1);     /* z = 10.0*/
```

See Also

ceil, ceilf

fmod, fmodf

floating-point modulus

Synopsis

```
#include <math.h>
double fmod (double x, double y);
float fmodf (float x, float y);
```

Description

The `fmod` and `fmodf` functions compute the floating-point remainder that results from dividing the first argument by the second argument.

The result is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, `fmod` and `fmodf` return zero.

Error Conditions

The `fmod` and `fmodf` functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;

y = fmod (5.0, 2.0);      /* y = 1.0 */
x = fmodf (4.0, 2.0);     /* x = 0.0 */
```

See Also

div, ldiv, modf, modff

free

deallocate memory

Synopsis

```
#include <stdlib.h>
void free (void *ptr);
```

Description

The `free` function deallocates a pointer previously allocated to a range of memory (by `calloc` or `malloc`) to the free memory heap. If the pointer was not previously allocated by `calloc`, `malloc`, `realloc`, `heap_calloc`, `heap_malloc`, or `heap_realloc`, the behavior is undefined.

The `free` function returns the allocated memory to the heap from which it was allocated.

Error Conditions

The `free` function does not return an error condition.

Example

```
#include <stdlib.h>
char *ptr;

ptr = malloc (10);     /* Allocate 10 words from heap */
free (ptr);            /* Return space to free heap */
```

See Also

calloc, heap_calloc, heap_free, heap_init, heap_lookup, heap_malloc, heap_realloc, heap_switch, malloc, realloc

frexp, frexpf

separate fraction and exponent

Synopsis

```
#include <math.h>
double frexp (double x, int *expptr);
float frexpf (float x, int *expptr);
```

Description

The `frexp` and `frexpf` functions separate a floating-point input into a normalized fraction and a (base 2) exponent. The functions return a fraction in the interval [$\frac{1}{2}$, 1) and store a power of 2 in the integer pointed to by the second argument. If the input is zero, then both the value stored and the value returned are zero.

Error Conditions

The `frexp` and `frexpf` functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;
int exponent;

y = frexp (2.0, &exponent);    /* y = 0.5, exponent = 2 */
x = frexpf (4.0, &exponent);   /* x = 0.5, exponent = 3 */
```

See Also

modf, modff

getenv

get string definition from operating system

Synopsis

```
#include <stdlib.h>
char *getenv (const char *name);
```

Description

The `getenv` function polls the operating system to see if a string is defined. There is no default operating system for the ADSP-21xxx, so `getenv` always returns NULL.

Error Conditions

The `getenv` function does not return an error condition.

Example

```
#include <stdlib.h>
char *ptr;

ptr = getenv ("ADI_DSP");   /* ptr = NULL */
```

See Also

system

heap_calloc

allocate and initialize memory in a heap

Synopsis

```
#include <stdlib.h>
void *heap_calloc(int heap_index, size_t nelem, size_t size);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The function allocates from the heap identified by `heap_index`, an array containing `nelem` elements of size `size`, and stores zeros in all bytes of the array. If successful, it returns a pointer to this array; otherwise, it returns a NULL pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see "Support for Multiple Heaps" on page 2-111.

Error Conditions

The `heap_calloc` function returns the NULL pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>

int main()
{
   int index,ok,prev;
   char *buf;

   /* Obtain the heap index for the user identifier 2 */
   index = heap_lookup(2);
```

```
    if (index < 0) {
        printf("Heap with user id of 2 not found\n");
        return 1;
    }
    /* initialize the heap so that it is ready for use */
    ok = heap_init(index);
    if (ok != 0) {
        printf("Heap failed to initialize, error= %d\n",ok);
        return 1;
    } else {
        printf("Heap initialised successfully\n");
    }
    /* allocate memory for 128 characters from heap 2 */
    buf = (char *)heap_calloc(index,128,sizeof(char));
    if (buf != 0) {
        free(buf); /* free can be used to release the memory */
    } else {
        printf("Unable to allocate from heap 2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_free, heap_init, heap_lookup, heap_malloc, heap_realloc, heap_switch, malloc, realloc

heap_free

return memory to a heap

Synopsis

```
#include <stdlib.h>
void heap_free(int heap_index, void *ptr);
```

Description

This function is an Analog Devices extension to the ANSI standard.

If `ptr` is not a NULL pointer, the function deallocates the object whose address is `ptr`; otherwise, it does nothing. The argument `heap_index` must be the index of the heap from which the object pointed to by `ptr` was originally allocated. If the object was not allocated from the specified heap, then the behavior is undefined.

The `heap_free` function is somewhat faster than `free`, but `free` must be used if the heap from which the object was allocated is not known with certainty.

For more information on creating multiple run-time heaps, see "Support for Multiple Heaps" on page 2-111.

Error Conditions

The `heap_free` function does not return an error condition.

Example

```
#include <stdlib.h>

int main()
{
   int index,ok,prev;
   char *buf;
```

```
    /* Obtain the heap index for the user identifier 2 */
    index = heap_lookup(2);
    if (index < 0) {
        printf("Heap with user id of 2 not found\n");
        return 1;
    }
    /* initialize the heap so that it is ready for use */
    ok = heap_init(index);
    if (ok != 0) {
        printf("Heap failed to initialize, error= %d\n",ok);
        return 1;
    } else {
        printf("Heap initialised successfully\n");
    }
    /* allocate memory for 128 characters from heap 2 */
    buf = (char *)heap_calloc(index,128,sizeof(char));
    if (buf != 0) {
        heap_free(index,buf); /* return the memory to heap 2 */
     } else {
        printf("Unable to allocate from heap 2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_init, heap_lookup, heap_malloc, heap_realloc, heap_switch, malloc, realloc

heap_init

initialize a heap

Synopsis

```
#include <stdlib.h>
int heap_init(int heap_index);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The function initializes or reinitializes a heap. All non-default heaps must be initialized before they can be used. This function may change the `start` or `size` field of the heap descriptor record for this heap in order to satisfy alignment requirements. In addition, if the heap is too small to contain the heap control block and at least one allocatable object, the `size` field will be set to 0, which indicates that the heap is not usable.

For more information on creating multiple run-time heaps, see "Support for Multiple Heaps" on page 2-111.

Error Conditions

The function returns 0 if successful; otherwise, it returns -2 if `heap_index` is out of range or -1 if the heap is too small to initialize.

Example

```
#include <stdlib.h>

int main()
{
   int index,ok;
   /* Obtain the heap index for the user identifier 1 */
   index = heap_lookup(1);
```

```
    if (index < 0) {
        printf("Heap with user id of 1 not found\n");
        return 1;
    }
    /* initialize the heap so that it is ready for use */
    ok = heap_init(index);
    if (ok != 0) {
        printf("Heap failed to initialize, error= %d\n",ok);
    } else {
        printf("Heap initialised successfully\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup, heap_malloc, heap_realloc, heap_switch, malloc, realloc

heap_lookup

obtain primary heap identifier

Synopsis

```
#include <stdlib.h>
int heap_lookup(int user_id);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The function returns the primary heap identifier of the heap with user identifier `user_id`, if there is such a heap; otherwise, -1 is returned. The primary heap identifier is the index of the heap descriptor record in the heap descriptor table. The user identifier for a heap is determined by a field in the heap descriptor record. The default heap always has user identifier 0.

For more information on multiple run-time heaps, see "Support for Multiple Heaps" on page 2-111.

Error Conditions

The function returns -1 if the specified user identifier was not found, otherwise it returns the primary head identifier of the specified heap.

Example

```
#include <stdlib.h>

int main()
{
   int index,ok;
   /* Obtain the heap index for the user identifier 1 */
   index = heap_lookup(1);
```

```
    if (index < 0) {
        printf("Heap with user id of 1 not found\n");
        return 1;
    }
    /* initialize the heap so that it is ready for use */
    ok = heap_init(index);
    if (ok != 0) {
        printf("Heap failed to initialize, error= %d\n",ok);
    } else {
        printf("Heap initialised successfully\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_malloc, heap_realloc, heap_switch, malloc, realloc

heap_malloc

allocate memory from a heap

Synopsis

```
#include <stdlib.h>
void *heap_malloc(int heap_index, size_t size);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The function allocates an object of size `size` from the heap identified by `heap_index`. It returns the address of the object if successful; otherwise, it returns a NULL pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

The block of memory is uninitialized. The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see "Support for Multiple Heaps" on page 2-111.

Error Conditions

The `heap_malloc` function returns the NULL pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>

int main()
{
   int index,ok,prev;
   char *buf;

   /* Obtain the heap index for the user identifier 2 */
   index = heap_lookup(2);
```

```
    if (index < 0) {
        printf("Heap with user id of 2 not found\n");
        return 1;
    }
    /* initialize the heap so that it is ready for use */
    ok = heap_init(index);
    if (ok != 0) {
        printf("Heap failed to initialize, error= %d\n",ok);
        return 1;
    } else {
        printf("Heap initialised successfully\n");
    }
    /* allocate memory for 128 characters from heap 2 */
    buf = (char *)heap_malloc(index,128);
    if (buf != 0) {
        free(buf); /* free can be used to release the memory */
    } else {
        printf("Unable to allocate from heap 2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_lookup, heap_realloc, heap_switch, malloc, realloc

heap_realloc

change memory allocation from a heap

Synopsis

```
#include <stdlib.h>
void *heap_realloc(int heap_index, void *ptr, size_t size);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The function allocates from the heap identified by `heap_index` an object of size `size`, obtaining initial stored values from the object whose address is `ptr`. It returns the address of the object if successful; otherwise, it returns a NULL pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

If `ptr` is not a NULL pointer, it must be the address of an existing object that you first allocate by calling `calloc`, `malloc`, `realloc`, `heap_calloc`, `heap_malloc`, or `heap_realloc`. The heap identified by `heap_index` must be the same as the heap from which the object was originally allocated; if it is not the same, the behavior is undefined. If the existing object is not larger than the newly allocated object, `heap_realloc` copies the entire existing object to the initial part of the allocated object. (The values stored in the remainder of the object are indeterminate.)

Otherwise, the function copies only the initial part of the existing object that fits in the allocated object. If `heap_realloc` succeeds in allocating a new object, it deallocates the existing object. Otherwise, the existing object is left unchanged.

If `ptr` is a NULL pointer, the values stored in the newly created object are indeterminate.

If `size` is 0 and `ptr` is not a NULL pointer, the object pointed to by `ptr` is deallocated and a NULL pointer is returned. However, if the object was originally allocated from a different heap from the heap identified by `heap_index`, the behavior is undefined.

The `heap_realloc` function is somewhat faster than realloc for resizing or deallocating an existing object, but realloc must be used if the heap from which the object was originally allocated is not known with certainty.

The allocated memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see "Support for Multiple Heaps" on page 2-111.

Error Conditions

The `heap_realloc` function returns the NULL pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>

int main()
{
   int index,ok,prev;
   char *buf,*upd;

   /* Obtain the heap index for the user identifier 2 */
   index = heap_lookup(2);
   if (index < 0) {
       printf("Heap with user id of 2 not found\n");
       return 1;
   }
   /* initialize the heap so that it is ready for use */
   ok = heap_init(index);
   if (ok != 0) {
       printf("Heap failed to initialize, error= %d\n",ok);
       return 1;
```

```
    } else {
        printf("Heap initialised successfully\n");
    }
    /* allocate memory for 128 characters from heap 2 */
    buf = (char *)heap_malloc(index,128);
    if (buf != 0) {
        strcpy(buf,"hello");
        /* change allocated size to 256 */
        upd = (char *)heap_realloc(index,buf,256);
        if (upd != 0) {
            printf("reallocated string for %s\n",upd);
            heap_free(index,upd); /* return to heap 2 */
        } else {
            free(buf); /* free can be used to release buf */
        }
    } else {
        printf("Unable to allocate from heap 2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_lookup, heap_malloc, heap_switch, malloc, realloc

heap_switch

set heap for dynamic memory allocation

Synopsis

```
#include <stdlib.h>
int heap_switch(int heap_index);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The function selects the current heap to be the heap identified by `heap_index`, which is the index of the heap's heap descriptor record. The heap index can be determined by using the `heap_lookup()` function. The default heap always has heap index 0.

Heaps other than the default heap must have been previously initialized with `heap_init` before `heap_switch` is called. If the call is not successful, the current heap is not changed.

The standard `malloc`, `calloc`, and `realloc` functions allocate new objects from the current heap. Thus, a successful call of `heap_switch` causes these functions to allocate from the specified heap.

The function returns the index of the previous heap, if the call was successful, and a negative number if it was not successful.

For more information on creating multiple run-time heaps, see "Support for Multiple Heaps" on page 2-111.

The `heap_switch` function is not available in multithreaded environments.

Error Conditions

The function returns the index of the previous heap, if the call was successful, unless there was no previous heap; in that case, it returns -1. It returns -2 if the size of the heap was too small to use. It returns -3 if `heap_index` was invalid (beyond the range of the heap descriptor table).

Example

```
#include <stdlib.h>

int main()
{
   int index,ok,prev;
   char *buf;

   /* Obtain the heap index for the user identifier 2 */
   index = heap_lookup(2);
   if (index < 0) {
       printf("Heap with user id of 2 not found\n");
       return 1;
   }
   /* initialize the heap so that it is ready for use */
   ok = heap_init(index);
   if (ok != 0) {
       printf("Heap failed to initialize, error= %d\n",ok);
       return 1;
   } else {
       printf("Heap initialised successfully\n");
   }
   prev = heap_switch(index);
   if (prev < 0) {
       printf("Failed to switch to heap 2, error = %d\n",prev);
       return 1;
    } else {
       buf = malloc(128); /* allocate buf from heap 2 */
       /* switch back to the previous heap */
       heap_switch(prev);
   }
   return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_lookup, heap_malloc, heap_realloc, malloc, realloc

interrupt

define interrupt handling

Synopsis

```
#include <signal.h>
void (*interrupt (int sig, void(*func)(int))) (int);
void (*interruptnsm (int sig, void(*func)(int))) (int);
void (*interruptf (int sig, void(*func)(int))) (int);
void (*interruptfnsm (int sig, void(*func)(int))) (int);
void (*interrupts (int sig, void(*func)(int))) (int);
void (*interruptsnsm (int sig, void(*func)(int))) (int);
void (*interruptcb (int sig, void(*func)(int))) (int);
void (*interruptcbnsm (int sig, void(*func)(int))) (int);
```

Description

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every interrupt `sig`; the `signal` function executes the function only once. The `func` argument must be one of the following that are listed in Table 3-10. The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

Table 3-10. Interrupt Handling

Func Value	Action
`SIG_DFL`	The default action is taken.
`SIG_IGN`	The signal is ignored.
Function address	The function pointed to by `func` is executed.

The function pointed to by `func` is executed each time the interrupt is received. The `interrupt` function must be called with the `SIG_IGN` argument to disable interrupt handling.

The differences between the functions `interrupt`, `interruptf`, `interrupts`, `interruptcb`, `interruptnsm`, `interruptfnsm`, `interruptsnsm`, and `interruptcbnsm` are discussed under "signal.h" on page 3-10.

Error Conditions

The `interrupt` function returns `SIG_ERR` and sets `errno` equal to `SIG_ERR` if the requested interrupt is not recognized.

Example

```
#include <signal.h>

interrupt (SIG_IRQ2, irq2_handler);
/* enable interrupt 2 whose handling routine is pointed to by
irq2_handler */

interrupt (SIG_IRQ2, SIG_IGN);
/* disable interrupt 2 */
```

See Also

raise, signal

isalnum

detect alphanumeric character

Synopsis

```
#include <ctype.h>
int isalnum (int c);
```

Description

The `isalnum` function determines if the argument is an alphanumeric character (`A-Z`, `a-z`, or `0-9`). If the argument is not alphanumeric, `isalnum` returns a zero. If the argument is alphanumeric, `isalnum` returns a nonzero value.

Error Conditions

The `isalnum` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%3s", isalnum (ch) ? "alphanumeric" : "");
    putchar ('\n');
}
```

See Also

isalpha, isdigit

isalpha

detect alphabetic character

Synopsis

```
#include <ctype.h>
int isalpha (int c);
```

Description

The `isalpha` function determines if the argument is an alphabetic character (`A-Z` or `a-z`). If the argument is not alphabetic, `isalpha` returns a zero. If the argument is alphabetic, `isalpha` returns a nonzero value.

Error Conditions

The `isalpha` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isalpha (ch) ? "alphabetic" : "");
    putchar ('\n');
}
```

See Also

isalnum, islower, isupper

iscntrl

detect control character

Synopsis

```
#include <ctype.h>
int iscntrl (int c);
```

Description

The `iscntrl` function determines if the argument is a control character (`0x00-0x1F` or `0x7F`). If the argument is not a control character, `iscntrl` returns a zero. If the argument is a control character, `iscntrl` returns a nonzero value.

Error Conditions

The `iscntrl` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", iscntrl (ch) ? "control" : "");
    putchar ('\n');
}
```

See Also

isalnum, isgraph

isdigit

detect decimal digit

Synopsis

```
#include <ctype.h>
int isdigit (int c);
```

Description

The `isdigit` function determines if the argument character is a decimal digit (`0-9`). If the argument is not a digit, `isdigit` returns a zero. If the argument is a digit, `isdigit` returns a nonzero value.

Error Conditions

The `isdigit` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isdigit (ch) ? "digit" : "");
    putchar ('\n');
}
```

See Also

isalnum, isxdigit

isgraph

detect printable character, not including white space

Synopsis

```
#include <ctype.h>
int isgraph (int c);
```

Description

The `isgraph` function determines if the argument is a printable character, not including space (`0x21-0x7e`). If the argument is not a printable character, `isgraph` returns a zero. If the argument is a printable character, `isgraph` returns a nonzero value.

Error Conditions

The `isgraph` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isgraph (ch) ? "graph" : "");
    putchar ('\n');
}
```

See Also

isalnum, iscntrl, isprint

islower

detect lowercase character

Synopsis

```
#include <ctype.h>
int islower (int c);
```

Description

The `islower` function determines if the argument is a lowercase character (`a-z`). If the argument is not lowercase, `islower` returns a zero. If the argument is lowercase, `islower` returns a nonzero value.

Error Conditions

The `islower` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", islower (ch) ? "lowercase" : "");
    putchar ('\n');
}
```

See Also

isalpha, isupper

isprint

detect printable character

Synopsis

```
#include <ctype.h>
int isprint (int c);
```

Description

The `isprint` function determines if the argument is a printable character (`0x20-0x7E`). If the argument is not a printable character, `isprint` returns a zero. If the argument is a printable character, `isprint` returns a nonzero value.

Error Conditions

The `isprint` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%3s", isprint (ch) ? "printable" : "");
    putchar ('\n');
}
```

See Also

isgraph, isspace

ispunct

detect punctuation character

Synopsis

```
#include <ctype.h>
int ispunct (int c);
```

Description

The `ispunct` function determines if the argument is a punctuation character. If the argument is not a punctuation character, `ispunct` returns a zero. If the argument is a punctuation character, `ispunct` returns a nonzero value.

Error Conditions

The `ispunct` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%3s", ispunct (ch) ? "punctuation" : "");
    putchar ('\n');
}
```

See Also

isalnum

isspace

detect whitespace character

Synopsis

```
#include <ctype.h>
int isspace (int c);
```

Description

The `isspace` function determines if the argument is a blank space character (`0x09-0x0D` or `0x20`). This includes space `( )`, form feed (`\f`), new line (`\n`), carriage return (`\r`), horizontal tab (`\t`) and vertical tab (`\v`). If the argument is not a blank space character, `isspace` returns a zero. If the argument is a blank space character, `isspace` returns a nonzero value.

Error Conditions

The `isspace` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isspace (ch) ? "space" : "");
    putchar ('\n');
}
```

See Also

iscntrl, isgraph

isupper

detect uppercase character

Synopsis

```
#include <ctype.h>
int isupper (int c);
```

Description

The `isupper` function determines if the argument is an uppercase character (`A-Z`). If the argument is not an uppercase character, `isupper` returns a zero. If the argument is an uppercase character, `isupper` returns a nonzero value.

Error Conditions

The `isupper` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isupper (ch) ? "uppercase" : "");
    putchar ('\n');
}
```

See Also

isalpha, islower

isxdigit

detect hexadecimal digit

Synopsis

```
#include <ctype.h>
int isxdigit (int c);
```

Description

The `isxdigit` function determines if the argument character is a hexadecimal digit character (`A-F`, `a-f`, or `0-9`). If the argument is not a hexadecimal digit, `isxdigit` returns a zero. If the argument is a hexadecimal digit, `isxdigit` returns a nonzero value.

Error Conditions

The `isxdigit` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isxdigit (ch) ? "hexadecimal" : "");
    putchar ('\n');
}
```

See Also

isalnum, isdigit

labs

absolute value

Synopsis

```
#include <stdlib.h>
long int labs (long int j);
```

Description

The `labs` function returns the absolute value of its integer argument.

Note: `labs (LONG_MIN) == LONG_MIN`.

Error Conditions

The `labs` function does not return an error condition.

Example

```
#include <stdlib.h>
long int j;

j = labs (-285128);   /* j = 285128 */
```

See Also

abs, fabs, fabsf

lavg

return mean of two values

Synopsis

```
#include <stdlib.h>
long int lavg (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lavg` function adds two arguments and divides the result by two. The `lavg` function is a built-in function which is implemented with an `Rn=(Rx+Ry)/2` instruction.

Error Conditions

The `lavg` function does not return an error code.

Example

```
#include <stdlib.h>
long int i;

i = lavg (10, 8);   /* returns 9 */
```

See Also

avg, favg, favgf

lclip

clip x by y

Synopsis

```
#include <stdlib.h>
long int lclip (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lclip` function returns its first argument if it is less than the absolute value of its second argument, otherwise it returns the absolute value of its second argument of the first is positive, or minus the absolute value if the first argument is negative. The `lclip` function is a built-in function which is implemented with an `Rn=CLIP Rx BY Ry` instruction.

Error Conditions

The `lclip` function does not return an error code.

Example

```
#include <stdlib.h>
long int i;

i = lclip (10, 8);   /* returns 8 */
i = lclip (8, 10);   /* returns 8 */
i = lclip (-10, 8);  /* returns -8 */
```

See Also

clip, fclip, fclipf

ldexp, ldexpf

multiply by power of 2

Synopsis

```
#include <math.h>
double ldexp (double x, int n);
float ldexpf (float x, int n);
```

Description

The `ldexp` and `ldexpf` functions return the value of the floating-point argument multiplied by 2^n. The `ldexp` and `ldexpf` functions add the value of `n` to the exponent of `x`.

Error Conditions

If the result overflows, `ldexp` and `ldexpf` return `HUGE_VAL` with the proper sign and set `errno` to `ERANGE`. If the result underflows, a zero is returned.

Example

```
#include <math.h>
double y;
float x;

y = ldexp (0.5, 2);    /* y = 2.0 */
x = ldexpf (1.0, 2);   /* x = 4.0 */
```

See Also

exp, expf, pow, powf

ldiv

division

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv (long int numer, long int denom);
```

Description

The `ldiv` function divides `numer` by `denom`, and returns a structure of type `ldiv_t`. The type `ldiv_t` is defined as:

```
typedef struct {
   long int quot;
   long int rem;
} ldiv_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if result is of type `ldiv_t`:

```
result.quot * denom + result.rem == numer
```

Error Conditions

If `denom` is zero, the behavior of the `ldiv` function is undefined.

Example

```
#include <stdlib.h>
ldiv_t result;

result = ldiv (7L, 2L);   /* result.quot = 3, result.rem = 1 */
```

See Also

div, fmod, fmodf

lmax

return larger of two values

Synopsis

```
#include <stdlib.h>
long int lmax (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lmax` function returns the larger of its two arguments. The `lmax` function is a built-in function which is implemented with an `Rn=MAX(Rx,Ry)` instruction.

Error Conditions

The `lmax` function does not return an error code.

Example

```
#include <stdlib.h>
long int i;

i = lmax (10, 8);   /* returns 10 */
```

See Also

fmax, fmaxf, fmin, fminf, lmax, lmin, max, min

lmin

return the smaller of two values

Synopsis

```
#include <stdlib.h>
long int lmin (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lmin` function returns the smaller of its two arguments. The `lmin` function is a built-in function which is implemented with an `Rn=MIN(Rx,Ry)` instruction.

Error Conditions

The `lmin` function does not return an error code.

Example

```
#include <stdlib.h>
long int i;

i = lmin (10, 8);   /* returns 8 */
```

See Also

fmax, fmaxf, fmin, fminf, lmax, max, min

localeconv

get pointer for formatting to current locale

Synopsis

```
#include <locale.h>
struct lconv *localeconv (void);
```

Description

The `localeconv` function returns a pointer to an object of type `struct lconv`. This pointer is used to set the components of the object with values used in formatting numeric quantities in the current locale.

With the exception of `decimal_point`, those members of the structure with type `char *` may use `" "` to indicate that a value is not available. Expected values are strings. Those members with type `char` may use `CHAR_MAX` to indicate that a value is not available. Expected values are non-negative numbers.

The program may not alter the structure pointed to by the return value but subsequent calls to `localeconv` may do so. Also, calls to `setlocale` with the category arguments of `LC_ALL`, `LC_MONETARY` and `LC_NUMERIC` may overwrite the structure.

Table 3-11. Members of the lconv Struct

Member	Description
`char *currency_symbol`	Currency symbol applicable to the locale
`char *decimal_point`	Used to format nonmonetary quantities
`char *grouping`	Used to indicate the number of digits in each nonmonetary grouping

Table 3-11. Members of the lconv Struct (Cont'd)

Member	Description
`char *int_curr_symbol`	Used as international currency symbol (ISO 4217:1987) for that particular locale plus the symbol used to separate the currency symbol from the monetary quantity
`char *mon_decimal_point`	Used for decimal point format monetary quantities
`char *mon_grouping`	Used to indicate the number of digits in each monetary grouping
`char *mon_thousands_sep`	Used to group monetary quantities prior to the decimal point
`char *negative_sign`	Used to indicate a negative monetary quantity
`char *positive_sign`	Used to indicate a positive monetary quantity
`char *thousands_sep`	Used to group nonmonetary quantities prior to the decimal point
`char frac_digits`	Number of digits displayed after the decimal point in monetary quantities in other than international format
`char int_frac_digits`	Number of digits displayed after the decimal point in international monetary quantities
`char p_cs_precedes`	If set to 1, the `currency_symbol` precedes the positive monetary quantity. If set to 0, the `currency_symbol` succeeds the positive monetary quantity.
`char n_cs_precedes`	If set to 1, the `currency_symbol` precedes the negative monetary quantity. If set to 0, the `currency_symbol` succeeds the negative monetary quantity.
`char n_sign_posn`	Indicates the positioning of `negative_sign` for monetary quantities.

Table 3-11. Members of the lconv Struct (Cont'd)

Member	Description
`char n_sep_by_space`	If set to 1, the `currency_symbol` is separated from the negative monetary quantity. If set to 0, the `currency_symbol` is not separated from the negative monetary quantity.
`char p_sep_by_space`	If set to 1, the `currency_symbol` is separated from the positive monetary quantity. If set to 0, the `currency_symbol` is not separated from the positive monetary quantity.

For `grouping` and `non_grouping` an element of `CHAR_MAX` indicates that no further grouping will be performed, a 0 indicates that the previous element should be used to group the remaining digits and any other integer value is used as the number of digits in the current grouping.

The definitions of the values for `p_sign_posn` and `n_sign_posn` are as follows:

- parentheses surround `currency_symbol` and quantity
- sign string precedes `currency_symbol` and quantity
- sign string succeeds `currency_symbol` and quantity
- sign string immediately precedes `currency_symbol`
- sign string immediately succeeds `currency_symbol`

Error Conditions

The localeconv function does not return an error condition.

Example

```
#include <locale.h>
struct lconv *c_locale;

c_locale = localeconv (); /* Only the C locale is */
                          /* currently supported */
```

See Also

setlocale

log, logf

natural logarithm

Synopsis

```
#include <math.h>
double log (double x);
float logf (float x);
```

Description

The `log` and `logf` functions compute the natural (base e) logarithm of their argument.

The `log` and `logf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `log` and `logf` functions return zero and set `errno` to `EDOM` if the input value is zero or negative.

Example

```
#include <math.h>
double y;
float x;

y = log (1.0);          /* y = 0.0 */
x = logf (2.71828);     /* x = 1.0 */
```

See Also

exp, expf, log10, log10f

log10, log10f

base 10 logarithm

Synopsis

```
#include <math.h>
double log10 (double x);
float log10f (float x);
```

Description

The `log10` and `log10f` functions produce the base 10 logarithm of their argument.

The `log10` and `log10f` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `log10` and `log10f` functions indicate a domain error (set `errno` to `EDOM`) and return zero if the input is zero or negative.

Example

```
#include <math.h>
double y;
float x;

y = log10 (100.0);     /* y = 2.0 */
x = log10f (10.0);     /* x = 1.0 */
```

See Also

log, logf, pow, powf

longjmp

second return from `setjmp`

Synopsis

```
#include <setjmp.h>
void longjmp (jmp_buf env, int return_val);
```

Description

The `longjmp` function causes the program to execute a second return from the place where `setjmp (env.)` was called (in same `env` buffer).

The `longjmp` function takes as its arguments a jump buffer that contains the context at the time of the original `setjmp`. It also takes an integer, `return_val`, which `setjmp` returns if `return_val` is nonzero. Otherwise, `setjmp` returns a 1.

If `env` was not initialized through a previous call to `setjmp` or the function that called `setjmp` has since returned, the behavior is undefined. Also, automatic variables that are local to the original function calling `setjmp`, that do not have `volatile`-qualified type, and that have changed their value prior to the `longjmp` call, have indeterminate value.

Error Conditions

The `longjmp` function does not return an error condition.

Example

```
#include <setjmp.h>
#include <stdio.h>
#include <errno.h>
```

```
#include <stdlib.h>

jmp_buf env;
int res;

if ((res == setjmp(env)) != 0) {
    printf  ("Problem %d reported by func ()", res);
    exit (EXIT_FAILURE);
}
func ();

void func (void)
{
    if (errno != 0) {
        longjmp (env, errno);
    }
}
```

See Also

setjmp

malloc

allocate memory

Synopsis

```
#include <stdlib.h>
void *malloc (size_t size);
```

Description

The `malloc` function returns a pointer to a block of memory of length `size`. The block of memory is uninitialized.

The object is allocated from the current heap, which will be the default heap unless `heap_switch` has been called to change the current heap to an alternate heap.

Error Conditions

The `malloc` function returns a NULL pointer if it is unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *)malloc (10); /* ptr points to an   */
                          /* array of length 10 */
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_lookup, heap_malloc, heap_realloc, heap_switch, realloc

max

return larger of two values

Synopsis

```
#include <stdlib.h>
int max (int value1, int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `max` function returns the larger of its two arguments. The `max` function is a built-in function which is implemented with an `Rn=MAX(Rx,Ry)` instruction.

Error Conditions

The `max` function does not return an error code.

Example

```
#include <stdlib.h>
int i;

i = max (10, 8);   /* returns 10 */
```

See Also

fmax, fmaxf, fmin, fminf, lmax, lmin, min

memchr

find first occurrence of character

Synopsis

```
#include <string.h>
void *memchr (const void *s1, int c, size_t n);
```

Description

The `memchr` function compares the range of memory pointed to by `s1` with the input character `c` and returns a pointer to the first occurrence of `c`. A NULL pointer is returned if `c` does not occur in the first `n` characters.

Error Conditions

The `memchr` function does not return an error condition.

Example

```
#include <string.h>
char *ptr;

ptr = memchr ("TESTING", 'E', 7);
    /* ptr points to the E in TESTING */
```

See Also

strchr, strrchr

memcmp

compare objects

Synopsis

```
#include <string.h>
int memcmp (const void *s1, const void *s2, size_t n);
```

Description

The `memcmp` function compares the first `n` characters of the objects pointed to by `s1` and `s2`. It returns a positive value if the `s1` object is lexically greater than the `s2` object, a negative value if the `s2` object is lexically greater than the `s1` object, and a zero if the objects are the same.

Error Conditions

The `memcmp` function does not return an error condition.

Example

```
#include <string.h>
char string1 = "ABC";
char string2 = "BCD";
int result;

result = memcmp (string1, string2, 3);   /* result < 0 */
```

See Also

strcmp, strcoll, strncmp

memcpy

copy characters from one object to another

Synopsis

```
#include <string.h>
void *memcpy (void *s1, const void *s2, size_t n);
```

Description

The `memcpy` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The behavior of `memcpy` is undefined if the two objects overlap; see `memmove`.

The `memcpy` function returns the address of `s1`.

Error Conditions

The `memcpy` function does not return an error condition.

Example

```
#include <string.h>
char *a = "SRC";
char *b = "DEST";

memcpy (b, a, 3);   /* *b = "SRC" */
```

See Also

memmove, strcpy, strncpy

memmove

copy characters from one object to another

Synopsis

```
#include <string.h>
void *memmove (void *s1, const void *s2, size_t n);
```

Description

The `memmove` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is copied correctly even if the objects overlap.

The `memmove` function returns a pointer to `s1`.

Error Conditions

The `memmove` function does not return an error condition.

Example

```
#include <string.h>
char *ptr, *str = "ABCDE";

ptr = str + 2;
memmove (ptr, str, 5);   /* *ptr = "ABCDE" */
```

See Also

memcpy, strcpy, strncpy

memset

set range of memory to a character

Synopsis

```
#include <string.h>
void *memset (void *s1, int c, size_t n);
```

Description

The `memset` function sets a range of memory to the input character `c`. The first `n` characters of `s1` are set to `c`.

The `memset` function returns a pointer to `s1`.

Error Conditions

The `memset` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

memset (string1, '\0', 50);   /* set string1 to 0 */
```

See Also

memcpy, memmove

min

return smaller of two values

Synopsis

```
#include <stdlib.h>
int min (int value1, int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `min` function returns the smaller of its two arguments. The `min` function is a built-in function which is implemented with an `Rn=MIN(Rx,Ry)` instruction.

Error Conditions

The `min` function does not return an error code.

Example

```
#include <stdlib.h>
int i;

i = min (10, 8);   /* returns 8 */
```

See Also

fmax, fmaxf, fmin, fminf, lmax, lmin, max

modf, modff

separate integral and fractional parts

Synopsis

```
#include <math.h>
double modf (double x, double *intptr);
float modff (float x, float *intptr);
```

Description

The `modf` and `modff` functions separate the first argument into integral and fractional portions. The fractional portion is returned and the integral portion is stored in the object pointed to by `intptr`. The integral and fractional portions have the same sign as the input.

Error Conditions

The `modf` and `modff` functions do not return error conditions.

Example

```
#include <math.h>
double y, n;
float m, p;

y = modf (-12.345, &n);   /* y = -0.345, n = -12.0 */
m = modff (11.75, &p);    /* m = 0.75, p = 11.0 */
```

See Also

frexp, frexpf

pow, powf

raise to a power

Synopsis

```
#include <math.h>
double pow (double x, double y);
float powf (float x, float y);
```

Description

The `pow` and `powf` functions compute the value of the first argument raised to the power of the second argument.

The `pow` and `powf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

A domain error occurs if the first argument is negative and the second argument cannot be represented as an integer. If the first argument is zero, the second argument is less than or equal to zero and the result cannot be represented, `EDOM` is stored in `errno` and zero is returned.

Example

```
#include <math.h>
double z;
float x;

z = pow (4.0, 2.0);      /* z = 16.0 */
x = powf (4.0, 2.0);     /* x = 16.0 */
```

See Also

ldexp, ldexpf, exp, expf

qsort

quicksort

Synopsis

```
#include <stdlib.h>
void qsort (void *base, size_t nelem, size_t size,
            int (*compar) (const void *, const void *));
```

Description

The `qsort` function sorts an array of `nelem` objects, pointed to by `base`. The size of each object is specified by `size`.

The contents of the array are sorted into ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

- If two elements compare as equal, their order in the sorted array is unspecified. `qsort` executes a binary-search operation on a pre-sorted array.
- `base` points to the start of the array.
- `nelem` is the number of elements in the array.
- `size` is the size of each element of the array.
- `compar` is a pointer to a function that is called by `qsort` to compare two elements of the array. The function should return a value less than, equal to, or greater than zero, according to whether the first argument is less than, equal to, or greater than the second.

Example

```
#include <stdlib.h>
float a[10];

int compare_float (const void *a, const void *b)
{
    float aval = *(float *)a;
    float bval = *(float *)b;
    if (aval < bval)
        return -1;
    else if (aval = = bval)
        return 0;
    else
        return 1;
}
qsort (a, sizeof (a)/sizeof (a[0]), sizeof (a[0]), compare_float);
```

See Also

bsearch

raise

force a signal

Synopsis

```
#include <signal.h>
int raise (int sig);
int raisensm(int sig);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `raise` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the signals listed in either Table 3-8 on page 3-46 or Table 3-9 on page 3-48.

The `raise` function uses self-modifying code. If this is not suitable for your application, then use the `raisensm` function instead. The choice of function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

Error Conditions

The `raise` function returns a zero if successful or a nonzero value if it fails.

Example

```
#include <signal.h>

raise (SIG_IRQ2);   /* invoke the interrupt 2 handler */
```

See Also

interrupt, signal

rand

random number generator

Synopsis

```
#include <stdlib.h>
int rand (void);
```

Description

The `rand` function returns a pseudo-random integer value in the range [0, $2^{32} - 1$].

For this function, the measure of randomness is its periodicity, the number of values it is likely to generate before repeating a pattern. The output of the pseudo-random number generator has a period on the order of $2^{32} - 1$.

Error Conditions

The `rand` function does not return an error condition.

Example

```
#include <stdlib.h>
int i;

i = rand ();
```

See Also

srand

realloc

change memory allocation

Synopsis

```
#include <stdlib.h>
void *realloc (void *ptr, size_t size);
```

Description

The function `realloc` changes the memory allocation of the object pointed to by `ptr` to `size`. Initial values for the new object are taken from those in the object pointed to by `ptr`. If the size of the new object is greater than the size of the object pointed to by `ptr`, then the values in the newly allocated section are undefined. If `ptr` is a non-NULL pointer that was not allocated with `malloc` or `calloc`, the behavior is undefined. If `ptr` is a NULL pointer, `realloc` imitates `malloc`. If `size` is zero and `ptr` is not a NULL pointer, `realloc` imitates `free`.

If `ptr` is non-NULL then the object is re-allocated from the heap that the object was originally allocated from. If `ptr` is NULL, then the object is allocated from the current heap, which will be the default heap unless `heap_switch` has been called to change the current heap to an alternate heap.

Error Conditions

If memory cannot be allocated, `ptr` remains unchanged and `realloc` returns a NULL pointer.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *)malloc (10);        /* intervening code  */
ptr = (int *)realloc (ptr, 20);  /* the size is now 20*/
```

See Also

calloc, free, heap_calloc, heap_free, heap_init, heap_lookup, heap_malloc, heap_realloc, heap_switch, malloc

setjmp

label for external linkage

Synopsis

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

Description

The `setjmp` function is a label declared with external linkage. The `setjmp` label saves the calling environment in the `jmp_buf` argument.

When `setjmp` is called, it immediately returns with zero. The call, in effect, declares the label. If, at some later point, `longjmp` is called within the same `jmp_buf` argument, `setjmp` will return a non-zero value. The call to `longjmp` causes a transfer to the label declared with `setjmp`.

Error Conditions

The label `setjmp` does not return an error condition.

Example

See "longjmp" on page 3-105

See Also

longjmp

setlocale

set the current locale

Synopsis

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

Description

The `setlocale` function uses the parameters `category` and `locale` to select a current locale. The possible values for the `category` argument are those macros defined in `locale.h` beginning with "`LC_`". The only `locale` argument supported at this time is the "`C`" locale. If a NULL pointer is used for the `locale` argument, `setlocale` returns a pointer to a string which is the current locale for the given category argument. A subsequent call to `setlocale` with the same `category` argument and the string supplied by the previous `setlocale` call returns the locale to its original status. The string pointed to may not be altered by the program but may be overwritten by subsequent `setlocale` calls.

Error Conditions

The `setlocale` function does not return an error condition.

Example

```
#include <locale.h>

setlocale (LC_ALL, "C");
/* sets the locale to the "C" locale */
```

See Also

localeconv

signal

define signal handling

Synopsis

```
#include <signal.h>
void (*signal (int sig, void (*func)(int))) (int);
void (*signalnsm (int sig, void (*func)(int))) (int);
void (*signalf (int sig, void (*func)(int))) (int);
void (*signalfnsm (int sig, void (*func)(int))) (int);
void (*signals (int sig, void (*func)(int))) (int);
void (*signalsnsm (int sig, void (*func)(int))) (int);
void (*signalcb (int sig, void (*func)(int))) (int);
void (*signalcbnsm (int sig, void (*func)(int))) (int);
```

Description

The `signal`, `signalnsm`, `signalf`, `signalfnsm`, `signals`, `signalsnsm`, `signalcb`, or `signalcbnsm` functions determine how a signal that is received during program execution is handled. The specified `signal`, `signalnsm`, `signalf`, `signalfnsm`, `signals`, `signalsnsm`, `signalcb`, or `signalcbnsm` function causes the corresponding `interrupt`, `interruptnsm`, `interruptf`, `interruptfnsm`, `interrupts`, `interruptsnsm`, `interruptcb`, or `interruptcbnsm` dispatcher to be used when handling the interrupt.

The signal function returns the value of the previously installed interrupt or signal handler action. The `sig` argument must be one of the values that are listed in either Table 3-8 on page 3-46 or Table 3-9 on page 3-48. The signal function causes the receipt of the signal number `sig` to be handled in one of the ways listed in Table 3-10 on page 3-79. The function pointed to by `func` is executed once when the signal is received. Handling is then returned to the default state.

The differences between the actions taken by the supplied standard interrupt dispatchers, `interrupt`, `interruptnsm`, `interruptf`, `interruptfnsm`, `interrupts`, `interruptsnsm`, `interruptcb`, and `interruptcbnsm`, are discussed under "signal.h" on page 3-10.

Error Conditions

The `signal` function returns `SIG_ERR` and sets `errno` to `SIG_ERR` if it does not recognize the requested signal.

Example

```
#include <signal.h>

signal (SIG_IRQ2, irq2_handler);    /* enable interrupt 2 */
signal (SIG_IRQ2, SIG_IGN);         /* disable interrupt 2 */
```

See Also

interrupt, raise

sin, sinf

sine

Synopsis

```
#include <math.h>
double sin (double x);
float sinf (float x);
```

Description

The `sin` and `sinf` functions return the sine of `x`. The input is interpreted as radians; the output is in the range [-1, 1].

The `sin` and `sinf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range. Although the `sin` and `sinf` functions accept input over the entire floating-point range, the accuracy of the result decreases significantly for inputs greater than $\pi^{12}/2$.

Error Conditions

The `sin` and `sinf` functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;

y = sin (3.14159);      /* y = 0.0 */
x = sinf (3.14159);     /* x = 0.0 */
```

See Also

asin, asinf

sinh, sinhf

hyperbolic sine

Synopsis

```
#include <math.h>
double sinh (double x);
float sinhf (float x);
```

Description

The `sinh` and `sinhf` functions return the hyperbolic sine of `x`.

The `sinh` and `sinhf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

For input values greater than 2^{12}, the `sinh` and `sinhf` functions return `HUGE_VAL` and set `errno` to `ERANGE` to indicate overflow.

Example

```
#include <math.h>
double x, y;
float z, w

y = sinh (x);
z = sinhf (w);
```

See Also

cosh, coshf

sqrt, sqrtf

square root

Synopsis

```
#include <math.h>
double sqrt (double x);
float sqrtf (float x);
```

Description

The `sqrt` and `sqrtf` functions return the positive square root of `x`.

The `sqrt` and `sqrtf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `sqrt` and `sqrtf` functions return zero for negative input values and set `errno` to `EDOM` to indicate a domain error.

Example

```
#include <math.h>
double y;
float x;

y = sqrt (2.0);         /* y = 1.414..... */
x = sqrtf (2.0);        /* x = 1.414..... */
```

See Also

rsqrt, rsqrtf

srand

random number seed

Synopsis

```
#include <stdlib.h>
void srand (unsigned int seed);
```

Description

The `srand` function is used to set the seed value for the `rand` function. A particular seed value always produces the same sequence of pseudo-random numbers.

Error Conditions

The `srand` function does not return an error condition.

Example

```
#include <stdlib.h>

srand (22);
```

See Also

rand

strcat

concatenate strings

Synopsis

```
#include <string.h>
char *strcat (char *s1, const char *s2);
```

Description

The `strcat` function appends a copy of the NULL-terminated string pointed to by `s2` to the end of the NULL-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string, which is NULL-terminated. The behavior of `strcat` is undefined if the two strings overlap.

Error Conditions

The `strcat` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

string1[0] = 'A';
string1[1] = 'B';
string1[2] = '\0';
strcat (string1, "CD");   /* new string is "ABCD" */
```

See Also

strncat

strchr

find first occurrence of character in string

Synopsis

```
#include <string.h>
char *strchr (const char *s1, int c);
```

Description

The `strchr` function returns a pointer to the first location in `s1`, a NULL-terminated string, that contains the character `c`.

Error Conditions

The `strchr` function returns NULL if `c` is not part of the string.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr (ptr1, 'E');
 /* ptr2 points to the E in TESTING */
```

See Also

memchr, strrchr

strcmp

compare strings

Synopsis

```
#include <string.h>
int strcmp (const char *s1, const char *s2);
```

Description

The `strcmp` function lexicographically compares the NULL-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

The `strcmp` function does not return an error condition.

Example

```
#include <string.h>
char string1[50], string2[50];

if (strcmp (string1, string2))
    printf ("%s is different than %s \n", string1, string2);
```

See Also

memcmp, strncmp

strcoll

compare strings

Synopsis

```
#include <string.h>
int strcoll (const char *s1, const char *s2);
```

Description

The `strcoll` function compares the string pointed to by `s1` with the string pointed to by `s2`. The comparison is based on the locale macro, `LC_COLLATE`. Because only the C locale is defined in the ADSP-21xxx environment, the `strcoll` function is identical to the `strcmp` function. The function returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

The `strcoll` function does not return an error condition.

Example

```
#include <string.h>
char string1[50], string2[50];

if (strcoll (string1, string2))
    printf ("%s is different than %s \n", string1, string2);
```

See Also

strcmp, strncmp

strcpy

copy from one string to another

Synopsis

```
#include <string.h>
char *strcpy (char *s1, const char *s2);
```

Description

The `strcpy` function copies the NULL-terminated string pointed to by `s2` into the space pointed to by `s1`. Memory allocated for `s1` must be large enough to hold `s2`, plus one space for the NULL character ('`\0`'). The behavior of `strcpy` is undefined if the two objects overlap or if `s1` is not large enough. The `strcpy` function returns the new `s1`.

Error Conditions

The `strcpy` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

strcpy (string1, "SOMEFUN");
/* SOMEFUN is copied into string1 */
```

See Also

memcpy, memmove, strncpy

strcspn

length of character segment in one string but not the other

Synopsis

```
#include <string.h>
size_t strcspn (const char *s1, const char *s2);
```

Description

The function `strcspn` returns the length of the initial segment of `s1` which consists entirely of characters not in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

The `strcspn` function does not return an error condition.

Example

```
#include <string.h>
char *ptr1, *ptr2;
size_t len;

ptr1 = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptr1, ptr2);  /* len = 2 */
```

See Also

strlen, strspn

strerror

get string containing error message

Synopsis

```
#include <string.h>
char *strerror (int errnum);
```

Description

The function `strerror` returns a pointer to a string containing an error message by mapping the number in `errnum` to that string. Only one error is currently defined in the ADSP-21xxx C environment.

Error Conditions

The `strerror` function does not return an error condition.

Example

```
#include <string.h>
char *ptr1;

ptr1 = strerror (1);
```

See Also

No references to this function.

strlen

string length

Synopsis

```
#include <string.h>
size_t strlen (const char *s1);
```

Description

The `strlen` function returns the length of the NULL-terminated string pointed to by `s1` (not including the NULL).

Error Conditions

The `strlen` function does not return an error condition.

Example

```
#include <string.h>
size_t len;

len = strlen ("SOMEFUN");   /* len = 7 */
```

See Also

strcspn, strspn

strncat

concatenate characters from one string to another

Synopsis

```
#include <string.h>
char *strncat (char *s1, const char *s2, size_t n);
```

Description

The `strncat` function appends a copy of up to `n` characters in the NULL-terminated string pointed to by `s2` to the end of the NULL-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string. The behavior of strncat is undefined if the two strings overlap. The new `s1` string is terminated with a NULL (`'\0'`).

Error Conditions

The `strncat` function does not return an error condition.

Example

```
#include <string.h>
char string1[50], *ptr;

string1[0] = '\0';
strncat (string1, "MOREFUN", 4);
/* string1 equals "MORE" */
```

See Also

strcat

strncmp

compare characters in strings

Synopsis

```
#include <string.h>
int strncmp (const char *s1, const char *s2, size_t n);
```

Description

The `strncmp` function lexicographically compares up to `n` characters of the NULL-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

The `strncmp` function does not return an error condition.

Example

```
#include <string.h>
char *ptr1;

ptr1 = "TEST1";
if (strncmp (ptr1, "TEST", 4) == 0)
    printf ("%s starts with TEST\n", ptr1);
```

See Also

memcmp, strcmp

strncpy

copy characters from one string to another

Synopsis

```
#include <string.h>
char *strncpy (char *s1, const char *s2, size_t n);
```

Description

The `strncpy` function copies up to `n` characters of the NULL-terminated string pointed to by `s2` into the space pointed to by `s1`. If the last character copied from `s2` is not a NULL, the result will not end with a NULL. The behavior of `strncpy` is undefined if the two objects overlap. The `strncpy` function returns the new `s1`.

If the `s2` string contains fewer than `n` characters, the `s1` string is padded with the NULL character until all `n` characters have been written.

Error Conditions

The `strncpy` function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

strncpy (string1, "MOREFUN", 4);
/* MORE is copied into string1 */
string1[4] = '\0';   /* must NULL-terminate string1 */
```

See Also

memcpy, memmove, strcpy

strpbrk

find character match in two strings

Synopsis

```
#include <string.h>
char *strpbrk (const char *s1, const char *s2);
```

Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

In the event that no character in `s1` matches any in `s2`, a NULL pointer is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2, *ptr3;

ptr1 = "TESTING";
ptr2 = "SHOP"
ptr3 = strpbrk (ptr1, ptr2);
/* ptr3 points to the S in TESTING */
```

See Also

strcmp, strncmp

strrchr

find last occurrence of character in string

Synopsis

```
#include <string.h>
char *strrchr (const char *s1, int c);
```

Description

The `strrchr` function returns a pointer to the last occurrence of character `c` in the NULL-terminated input string `s1`.

Error Conditions

The `strrchr` function returns a NULL pointer if `c` is not found.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strrchr (ptr1, 'T');
/* ptr2 points to the second T of TESTING */
```

See Also

memchr, strchr

strspn

length of segment of characters in both strings

Synopsis

```
#include <string.h>
size_t strspn (const char *s1, const char *s2);
```

Description

The `strspn` function returns the length of the initial segment of `s1` which consists entirely of characters in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

The `strspn` function does not return an error condition.

Example

```
#include <string.h>
size_t len;
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = "ERST";
len = strspn (ptr1, ptr2);     /* len = 4 */
```

See Also

strcspn, strlen

strstr

find string within string

Synopsis

```
#include <string.h>
char *strstr (const char *s1, const char *s2);
```

Description

The `strstr` function returns a pointer to the first occurrence in the string pointed to by `s1` of the characters in the string pointed to by `s2`. This excludes the terminating NULL character in `s1`.

Error Conditions

If the string is not found, `strstr` returns a NULL pointer. If `s2` points to a string of zero length, `s1` is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strstr (ptr1, "E");
 /* ptr2 points to the E in TESTING */
```

See Also

strchr

strtod

convert string to a double

Synopsis

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr);
```

Description

The `strtod` function returns the value that was represented by the string `nptr` as a `double`. The `strtod` function expects `nptr` to point to a string of the following form:

[*whitespace*] [*sign*] [*digits*] [.*digits*] [{d | D | e | E}[*sign*]*digits*]

The *whitespace* may consist of space and tab characters, which are ignored; *sign* is either plus (`+`) or minus (`-`); and *digits* are one or more decimal digits. If no digits appear before the radix character (`.`), at least one must appear after the radix character. The decimal digits can be followed by an exponent, which consists of an introductory letter (`d`, `D`, `e`, or `E`) and an optionally signed integer. If neither an exponent part nor a radix character appears, a radix character is assumed to follow the last digit in the string. The first character that does not fit this form stops the scan.

If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtod` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, `0` is returned. `ERANGE` is stored in `errno` in the case of either overflow or underflow.

Example

```
#include <stdlib.h>

char *rem;
double dd;

dd = strtod ("2345.5E4 abc", &rem);
/* dd=2.3455E+7, rem=" abc" */
```

See Also

strtoul, atoi, atol

strtok

convert string to tokens

Synopsis

```
#include <string.h>
char *strtok (char *s1, const char *s2);
```

Description

The `strtok` function returns successive tokens from the string `s1`, where each token is delimited by characters from `s2`.

A call to `strtok` with `s1` not NULL returns a pointer to the first token in `s1`, where a token is a consecutive sequence of characters not in `s2`. `s1` is modified in place to insert a NULL character at the end of the token returned. If `s1` consists entirely of characters from `s2`, NULL is returned.

Subsequent calls to `strtok` with `s1` equal to NULL will return successive tokens from the same string. When the string contains no further tokens, NULL is returned. Each new call to `strtok` may use a new delimiter string, even if `s1` is NULL, in which case the remainder of the string is tokenized using the new delimiter characters.

Error Conditions

The `strtok` function returns a NULL pointer if there are no tokens remaining in the string.

Example

```
#include <string.h>
static char str[] = "a phrase to be tested, today";
char *t;

t = strtok (str, " ");  /* t points to "a" */
t = strtok (NULL, " "); /* t points to "phrase" */
t = strtok (NULL, ","); /* t points to "to be tested" */
t = strtok (NULL, "."); /* t points to " today" */
t = strtok (NULL, "."); /* t = NULL */
```

See Also

No references to this function.

strtol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long int strtol (const char *nptr, char **endptr, int base);
```

Description

The function `strtol` returns as a `long int` the value that was represented by the string `nptr`. If `endptr` is not a NULL pointer, `strtol` stores a pointer to the unconverted remainder in `*endptr`.

The `strtol` function breaks down the input into three sections: white space (as determined by `isspace`), the initial characters, and the unrecognized characters, including a terminating NULL character. The initial characters may be composed of an optional sign character, 0x or 0X if `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and their use is permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtol` function returns a zero if no conversion can be made and the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, positive or negative (as appropriate) `LONG_MAX` is returned. If the correct value results in an underflow, `LONG_MIN` is returned. `ERANGE` is stored in `errno` in the case of either overflow or underflow.

Example

```
#include <stdlib.h>
#define base 10

char *rem;
long int i;

i = strtol ("2345.5", &rem, base);
/* i=2345, rem="5" */
```

See Also

atoi, atol, strtod, strtoul

strtoul

convert string to unsigned long integer

Synopsis

```
#include <stdlib.h>
unsigned long int strtoul (const char *nptr, char **endptr, int
base);
```

Description

The function `strtoul` returns as an `unsigned long int` the value represented by the string `nptr`. If `endptr` is not a NULL pointer, `strtoul` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoul` function breaks down the input into three sections:

- white space (as determined by `isspace`)
- the initial characters
- unrecognized characters including a terminating NULL character

The initial characters may be composed of an optional sign character (0x or 0X if `base` is 16) and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and their use is permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtoul` function returns a zero if no conversion can be made and the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, `ULONG_MAX` is returned. `ERANGE` is stored in `errno` in the case of overflow.

Example

```
#include <stdlib.h>
#define base 10

char *rem;
unsigned long int i;

i = strtoul ("2345.5", &rem, base);
/* i = 2345, rem = "5" */
```

See Also

atoi, atol, strtod, strtol

strxfrm

transform string using LC_COLLATE

Synopsis

```
#include <string.h>
size_t strxfrm (char *s1, const char *s2, size_t n);
```

Description

The strxfrm function transforms the string pointed to by s2 using the locale specific category LC_COLLATE. (See setlocale). It places the result in the array pointed to by s1.

The transformation is such that if s1 and s2 were transformed and used as arguments to strcmp, the result would be identical to the result derived from strcoll using s1 and s2 as arguments. However, since only C locale is implemented, this function does not perform any transformations other than the number of characters.

The string stored in the array pointed to by s1 is never more than n characters including the terminating NULL character. strxfrm returns 1. If this returned value is n or greater, the result stored in the array pointed to by s1 is indeterminate. s1 can be a NULL pointer if n is zero.

Error Conditions

The strxfrm function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

strxfrm (string1, "SOMEFUN", 49);
/* SOMEFUN is copied into string1 */
```

See Also

setlocale, strcmp, strcoll

system

send string to operating system

Synopsis

```
#include <stdlib.h>
int system (const char *string);
```

Description

The `system` function normally sends a string to the operating system. In the context of the ADSP-21xxx run-time environment, `system` always returns zero.

Error Conditions

The `system` function does not return an error condition.

Example

```
#include <stdlib.h>

system ("string");   /* always returns zero */
```

See Also

getenv

tan, tanf

tangent

Synopsis

```
#include <math.h>
double tan (double x);
float tanf (float x);
```

Description

The `tan` and `tanf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `tan` and `tanf` functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;

y = tan (3.14159/4.0);    /* y = 1.0 */
x = tanf (3.14159/4.0);   /* x = 1.0 */
```

See Also

atan, atanf

tanh, tanhf

hyperbolic tangent

Synopsis

```
#include <math.h>
double tanh (double x);
float tanhf (float x);
```

Description

The `tanh` and `tanhf` functions return a value that is accurate to 20 bits of the mantissa. This accuracy corresponds to a maximum relative error of 2^{-20} over its input range.

Error Conditions

The `tanh` and `tanhf` functions do not return an error condition.

Example

```
#include <math.h>
double x, y;
float z, w;

y = tanh (x);
z = tanhf (w);
```

See Also

cosh, coshf, sinh, sinhf

tolower

convert from uppercase to lowercase

Synopsis

```
#include <ctype.h>
int tolower (int c);
```

Description

The `tolower` function converts the input character to lowercase if it is uppercase; otherwise, it returns the character.

Error Conditions

The `tolower` function does not return an error condition.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (isupper (ch))
        printf ("tolower=%#04x", tolower (ch));
    putchar ('\n');
}
```

See Also

islower, isupper

toupper

convert from lowercase to uppercase

Synopsis

```
#include <ctype.h>
int toupper (int c);
```

Description

The `toupper` function converts the input character to uppercase if it is in lowercase; otherwise, it returns the character.

Error Conditions

The `toupper` function does not return an error condition.

Example

```
#include <ctype.h>
int ch;

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (islower (ch))
        printf ("toupper=%#04x", toupper (ch));
    putchar ('\n');
}
```

See Also

islower, isupper

va_arg

get next argument in variable-length list of arguments

Synopsis

```
#include <stdarg.h>
void va_arg (va_list ap, type);
```

Description

The `va_arg` macro is used to walk through the variable length list of arguments to a function.

After starting to process a variable-length list of arguments with `va_start`, call `va_arg` with the same `va_list` variable to extract arguments from the list. Each call to `va_arg` returns a new argument from the list.

Substitute a `type` name corresponding to the type of the next argument for the `type` parameter in each call to `va_arg`. After processing the list, call `va_end`.

The header file `stdarg.h` defines a pointer type called `va_list` that is used to access the list of variable arguments.

The function calling `va_arg` is responsible for determining the number and types of arguments in the list. It needs this information to determine how many times to call `va_arg` and what to pass for the type parameter each time. There are several common ways for a function to determine this type of information. The standard C `printf` function reads its first argument looking for %-sequences to determine the number and types of its extra arguments. In the example below, all of the arguments are of the same type (`char*`), and a termination value (NULL) is used to indicate the end of the argument list. Other methods are also possible.

If a call to `va_arg` is made after all arguments have been processed, or if `va_arg` is called with a type parameter that is different from the type of the next argument in the list, the behavior of `va_arg` is undefined.

Error Conditions

The `va_arg` macro does not return an error condition.

Example

```
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>

char *concat(char *s1, ...)
{
    int len = 0;
    char *result;
    char *s;
    va_list ap;

    va_start (ap, s1);
    s = s1;
    while (s) {
        len + = strlen (s);
        s = va_arg (ap, char *);
    }
    va_end (ap);

    result = malloc (len + 7);
    if  (!result)
        return result;
    *result = ' ';
    va_start (ap, s1);
    s = s1;
    while (s) {
        strcat (result, s);
        s = va_arg (ap, char *);
    }
    va_end (ap);
    return result;
 }
```

See Also

va_start, va_end

va_end

finish variable-length argument list processing

Synopsis

```
#include <stdarg.h>
void va_end (va_list ap);
```

Description

The `va_end` can only be invoked after the `va_start` macro. A call to `va_end` concludes the processing of a variable-length list of arguments that was begun by `va_start`.

Error Conditions

The `va_end` macro does not return an error condition.

Example

See "va_arg" on page 3-161

See Also

va_arg, va_start

va_start

initialize the variable-length argument list processing

Synopsis

```
#include <stdarg.h>
void va_start (va_list ap, parmN);
```

Description

The `va_start` macro is used in a function declared to take a variable number of arguments to start processing those variable arguments. The first argument to `va_start` should be a variable of type `va_list`, for use by `va_arg` in walking through the arguments. The second argument should be the last named parameter to the variable argument function. The `va_start` macro must be invoked before either the `va_arg` or `va_end` macro can be invoked.

Error Conditions

The `va_start` macro does not return an error condition.

Example

See "va_arg" on page 3-161

See Also

va_arg, va_end